

Performance Profiling and Optimization

on the SGI Origins

Sherry Chang
Scientific Consultant
Numerical Aerospace Simulation Facility
NASA Ames Research Center
MS 258-6
Moffett Field, CA 94035-1000

<http://www.nas.nasa.gov/~schang>

schang@nas.nasa.gov

Table of Contents

- [UNDERSTANDING THE ORIGINS](#)
 - [Distributed Memory Multi-Processors](#)
 - [Shared Memory Multi-Processors](#)
 - [Uniform Memory Access Model](#)
 - [Non-Uniform Memory Access Model](#)
 - [The Origin 2000 System](#)
 - [Node Board](#)
 - [CrayLink Interconnect](#)
 - [I/O](#)
 - [The Origins at NAS](#)
 - [The R10000/R12000 Processors](#)
 - [Instruction Latency](#)
 - [Memory Hierarchy](#)
 - [TLB, Virtual Memory and Physical Memory](#)
 - [Role of TLB in Memory Referencing](#)
 - [L1 and L2 Caches](#)
 - [Two-Way Set Associativity](#)
 - [The Situations that Cache Latency Applies](#)
 - [Principles of Good Cache Use](#)
 - [Memory Access Latency](#)
- [PERFORMANCE PROFILING](#)
 - [Available Tools](#)
 - [time, timex, ssusage, etime, dtime, second, timef](#)
 - [Perfex \(Hardware Performance Counter\)](#)
 - [The 32 R10000 Events](#)
 - [The 32 R12000 Events](#)
 - [perfex command](#)
 - [libperfex](#)
 - [SpeedShop - ssrun, prof and pixie](#)
 - [ssrun - SpeedShop Experiment](#)
 - [prof - SpeedShop Report Generation](#)
 - [pixie](#)
 - [How to Proceed](#)
 - [Measure the Time Performance of the Entire Code](#)
 - [Find Out the Sections/Routines that Use the Most Time](#)
 - [Find Out How Many Times Each Routine is Called and If Its Performance is Optimal](#)
 - [Find Out What Causes the Code not to Perform Well](#)
 - [For a Specific Hardware Performance Counter, Find Out Which Routines Contribute the Most](#)
- [SINGLE-CPU OPTIMIZATION TECHNIQUES](#)
 - [Sources of Performance Problems](#)
 - [Useful Techniques](#)
 - [Replace Division by Multiplication](#)
 - [Increase Page Size to Reduce TLB Miss and Page Fault](#)
 - [Interchange Loops to Improve Cache Utilization](#)
 - [Group Data Used at the Same Time to Reduce Traffic to Cache](#)
 - [Remove Cache Trashing by Re-dimensioning Array Sizes not to be Power of Two](#)
 - [Compiler Assisted Optimization](#)
 - [Useful Files Generated by Compiler](#)
 - [Optimization Option Groups](#)
 - [Basic Optimization Levels](#)
 - [Software Pipelining](#)
 - [Loop Nest Optimization](#)
 - [Array Padding](#)
 - [Loop Interchange](#)
 - [Outer Loop Unrolling](#)
 - [Cache Blocking](#)
 - [Loop Fusion](#)
 - [Loop Fission](#)
 - [Prefetching](#)
 - [Gather-Scatter](#)
 - [Vector Intrinsics](#)
 - [Interprocedural Analysis](#)
 - [Inlining](#)
 - [Miscellaneous Compiler Optimizations](#)
 - [Using Tuned Libraries for Optimizaiton](#)
 - [libm](#)
 - [libfastm](#)
 - [libcomplib.sgimath](#)
 - [libscsl](#)
- [Acknowledgement](#)

Links to Examples in this Document

- [Distributed memory parallel processors](#)
- [Shared memory parallel processors \(UMA\)](#)
- [Shared memory parallel processors \(NUMA\)](#)
- [Determination of the Cache Lines for a Datum](#)
- [L2 Cache Trashing among 3 Data with Identical Middle Address Bits](#)
- [Time profiling using etime](#)
- [Time profiling using dtime](#)
- [To find the cost \(in cycles\) for 1 occurrence of each event on the 250MHz R10000 hopper and steger](#)
- [To find the cost \(in cycles\) for 1 occurrence of each event on the 400MHz R12000 lomax](#)
- [Use perfex -a -x -y to find the cause of bad performance](#)
- [Use libperfex to measure performance](#)
- [PBS script for ssrun](#)
- [ssrun output](#)
- [ssrun -usertime output analyzed with prof -b -q 8 -dsolist](#)
- [ssrun -fpcsamp output analyzed with prof -l](#)
- [ssrun -fpcsamp output analyzed with prof -h](#)
- [ssrun -ideal output analyzed with prof -b -basicblock -q 8](#)
- [ssrun -dsc -hwc output analyzed with prof -l](#)
- [Traces floating point with ssrun -fpe and analyzed with prof](#)

- [Use pixie directly to count the execution frequency of each basic block](#)
- [Replace Division by Multiplication](#)
- [Increase Page Size to Reduce TLB Miss and Page Fault](#)
- [Interchange Loops to Improve Cache Utilization](#)
- [Group Data Used at the Same Time to Reduce Traffic to Cache](#)
- [Remove Cache Trashing by Re-dimensioning Array Sizes not to be Power of Two](#)
- [Compiler Assisted Optimization](#)
- [Software Pipelining - Instruction Scheduling](#)
- [Obtain Software Pipelining Messages](#)
- [Improve Cache Utilization with Cache Blocking](#)
- [Use Vector Intrinsics for Better Performance](#)
- [Optimization with Inter-Procedural Analysis](#)
- [Inlining with -INLINE](#)

[Glossary](#)

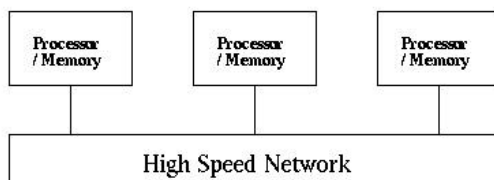
UNDERSTANDING THE ORIGINS

Distributed Memory Multi-Processors

In a multiprocessor configuration, each processor has fast access to its own local memory but no direct access to the memory of a different processor. (a processor does not know the address of a datum that resides in the memory of a different processor.) When data located in the memory of a different processor is needed, a message is sent via the inter-processor network.

Example : Distributed memory parallel processors

- Any networked computer (for example, PC and workstations) that can participate in clusters.
- Massively Parallel Processors that use fast distributed memory.



Shared Memory Multi-Processors

Memory in a multiprocessor configuration, usually RAM, which can be accessed by more than one processor, usually via a shared bus or network.

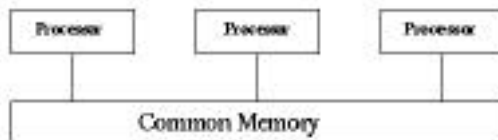
Two of the memory access models for shared memory machines:

● Uniform Memory Access (UMA) model:

In the UMA model, the memory is shared uniformly amongst all processors. It allows each processor equal access to all memory locations. The memory in UMA architecture is typically implemented in a central location with the processors acquiring access across a high-speed interconnection mechanism such as a bus or crossbar switch.

Example :

The Cray C90 machine

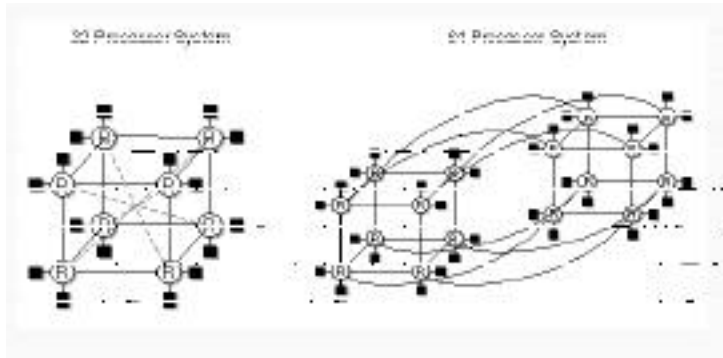


● Non-Uniform Memory Access (NUMA) model:

In the NUMA model the memory is physically distributed amongst the processors, but maintains a global address space. (a processor knows the address of a datum that resides in the memory of a different processor) The memory is still shared, but access time will differ depending on whether the requested memory address is local or remote to the requesting processor. A remote memory access requires a communication across the interconnection network which links the processors and thus the distributed memory.

Example :

The SGI origins machines.



Note: Each black square in the above diagram represents a 'node'. Nodes are connected through Routers/Craylink interconnect.

The Origin 2000 System

An Origin2000 system has the following components:

- **Node Board**

- **Processor(s)**

1 or 2 R10000 or R12000 processors with on-chip L1 (level-1) cache

- **L2 cache**

1-MB, 4-MB, 8-MB or 16-MB, level-2 cache, external to each processors

- **Hub**

allows processors to talk to its local memory, I/O devices and other processors

- **Main Memory**

64MB upto 4GB

- **Directory Memory**

This supplementary memory is controlled by the Hub. The directory keeps information about the cache status of all memory within its node. (i.e., keep track of which node (either local or remote) has exclusive ownership of a cache line resides in the local node, and which node(s) has a copy of a cache line that resides in the local node.) This status information is used to provide scalable cache coherence, and to migrate data to a node that accesses it more frequently than the present node.

- **CrayLink Interconnect**

This is a collection of very high speed links and routers that is responsible for tying together the set of hubs that make up the system. The important attributes of CrayLink Interconnect are its low latency, scalable bandwidth, modularity, and fault tolerance.

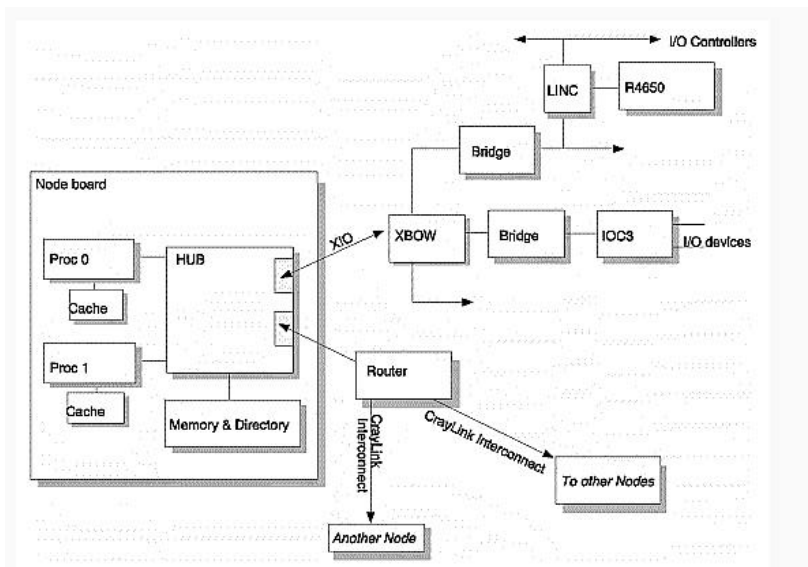
- **I/O Devices**

○ XIO and Crossbow Interfaces

These are the internal I/O interfaces originating in each Hub and terminating on the targeted I/O controller. XIO (XIO card) uses the same physical link technology as CrayLink Interconnect, but uses a protocol optimized for I/O traffic. The Crossbow ASIC is a crossbar routing chip responsible for connecting two nodes to up to six I/O controllers.

- **I/O Controllers**

Origin2000 supports a number of high-speed I/O interfaces, including Fast, Wide SCSI (a kind of bus), Fibrechannel, 100BASE-Tx, ATM, and HIPPI-Serial. Internally, these controllers are added through XIO cards, which have an embedded PCI-32 or PCI-64 bus. Thus, in Origin2000 I/O performance is added one bus at a time.



Suggested Reading:

[Application Programmer's I/C](#) **Figure 1-14 : Block Diagram of an Origin2000 System**

The Origin Machines at NAS

Here is a list of resources of the origin machines available for users:

Hostname	Turing	Fermi	Hopper	Steger	Lomax
Processor					
CPU	R10000	R10000	R10000	R10000	R12000
CPU-Clock	195MHz	195MHz	250MHz	250MHz	400MHz
NCPUS/Node	2	2	2	2	2
# of Nodes	12	4	32	128	256
NCPUS	24	8	64	256	512
Memory					
Local Memory/Node	512MB	250MB	512MB	512MB	768MB
Total Memory	7GB	1GB	16GB	64GB	192GB
Free Memory/Node	~490MB	N/A	~490MB	~490MB	~700MB
L1 Cache Size	32KB	32KB	32KB	32KB	32KB
L2 Cache Size	4MB	4MB	4MB	4MB	8MB
Page Size	16KB	16KB	16KB	16KB	16KB
I/O					
/u/userid	N/A	300MB	N/A	N/A	N/A
/scratch1	100GB	N/A	200GB	400GB	~800GB
/scratch2	100GB	N/A	200GB	400GB	~500GB
Function	front-end	server	compute	compute	compute

Note : 4 of the 12 nodes in **turing** contain ~760MB of memory each, the remaining 8 nodes contain 512MB of memory each.

For further information, refer to the configuration diagram for each machine. <http://in.nas.nasa.gov/Publications/ConfigDiags/>

The R10000/R12000 Processors

The MIPS R10000 and R12000 processors were designed with a few characteristics to achieve optimum performance.

● Four-Way Superscalar Architecture

Four-way	can fetch and decode 4 instructions per cycle
Superscalar	has enough independent, pipelined execution units that can complete more than one instruction per clock cycle

The execution units:

2 integer units	for address computation, arithmetic and logical operations on integers
2 floating point units	1 adder and 1 multiplier
1 load/store unit	for managing memory access

● Out-of-Order Execution

Out-of-Order	CPU dynamically executes instructions as their operands become available and thus instructions may not be executed in a predictable, sequential order
--------------	---

3 instruction queues (1 integer queue, 1 floating point queue and 1 load/store queue) are used to select which instructions to issue dynamically to the execution units. Up to 32 instructions can be in progress simultaneously.

The result generated out of order is temporary until all previous instructions have been successfully completed. At that time, the instruction executed out of order is 'graduated'.

Out-of-Order Execution helps to hide memory latency.

● Branch Prediction

Branch Prediction	In the case that the input operand of a conditional branch is not ready, CPU predicts which way the branch will go and executes the instructions speculatively along the path
-------------------	---

The CPU may speculate on the direction of branches nested four deep. (up to 4 outstanding branch predictions)

Branch prediction also helps to hide memory latency.

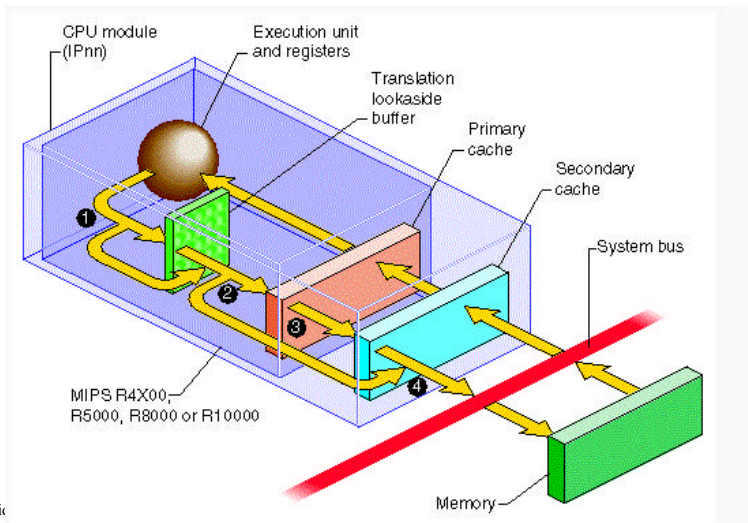
Instruction Latency

Load/Store	Latency (in clock periods)
Load (primary cache hit, int/FP)	2/3
Integer	
add, sub, logic ops, shift, branches	1
multiply (32-bits, signed)	5/6
multiply (64-bits, signed)	9/10
divide (32-bit/64-bit)	35/67
Floating Point	
add, compare, convert	2
multiply	2
multiply-add (bypass/not)	2/4
divide, reciprocal (single/double)	12/19
sqrt (single/double)	18/33
reciprocal sqrt (single/double)	30/52

Memory Hierarchy

- registers
- L1 instruction cache and L1 data cache
- L2 unified (instruction and data) cache
- local and remote memory
- disk

The CPU can make use only of data in registers, and it can load data into registers only from the primary cache. So data must be brought into the primary cache before it can be used in calculations. The primary cache can obtain data only from the secondary cache, so all data in the primary cache is simultaneously resident in the secondary cache. The secondary cache obtains its data from main memory.



What is "Translation Lookaside Buffer"?

Figure 1-1 CPU Access to Memory

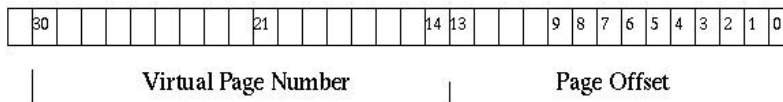
TLB, Virtual Memory :

The origins at NAS are so called 'Virtual Memory' machines which allow (1) the operating system to freely partition the physical memory of the system among the running processes while presenting a uniform virtual address space to all jobs and (2) processes to access address space much larger than the physical memory space.

Virtual memory is divided into pages. Thus, a page is the smallest continuous memory that the operating system can allocate to your program. For the origins at NAS, the default page size is set to be 16KB. However, this page size can be changed when necessary.

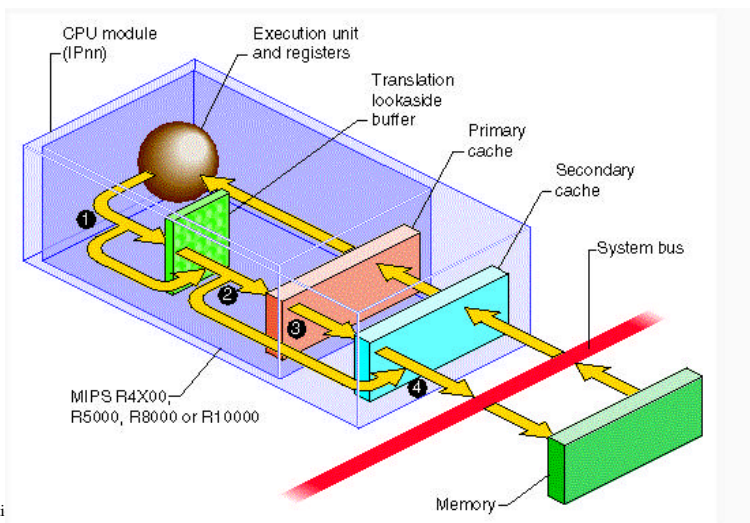
Each virtual address is composed of a virtual page number (the most significant bits) and an offset within the page (the N least significant bits, in the case of a page size 16KB, $N = 14$). In translating the virtual address to a physical address, the offset is left unchanged and the virtual page number is mapped to a physical page number. This is recombined with the offset to give a physical address (i.e., a location in physical memory).

For example, for 32-bit virtual address with a page size of 16KB, bits 13:0 represent the offset within a page and bits 30:14 represent a virtual page number from the 2^{*17} pages in the virtual segment.



The operating system translates between the virtual addresses your programs use and the physical addresses that the hardware requires. These translations are done for every memory access, so, to keep their overhead low, the 64 most recently used page addresses are cached in the translation lookaside buffer (TLB). This allows virtual-to-physical translation to be carried out with zero latency, in hardware - for those 64 pages.

Role of TLB in Memory Referencing



When a virtual memory location containing the **page number** occurred. Only then must the one of the TLB registers. The

Figure 1-1 CPU Access to Memory

he Translation Lookaside Buffer for an entry try for the referenced page in TLB, a TLB miss has physical address in a memory table and loads it into

Thus the TLB acts as a cache for frequently-used page addresses. The virtual memory of a program is usually much larger than 64 pages. The most recently used pages of memory (hundreds or thousands of them) are retained in physical memory, but depending on program activity and total system use, some virtual pages can be copied to disk and removed from memory.

A 'minor page fault' is said to occur when a program suffers a TLB miss but the referenced page is found in the physical memory by the operating system. If the operating system discovers that the referenced page is not presently in physical memory, a 'major page fault' has occurred and the program is further delayed while the page is read from disk.

L1 and L2 Caches

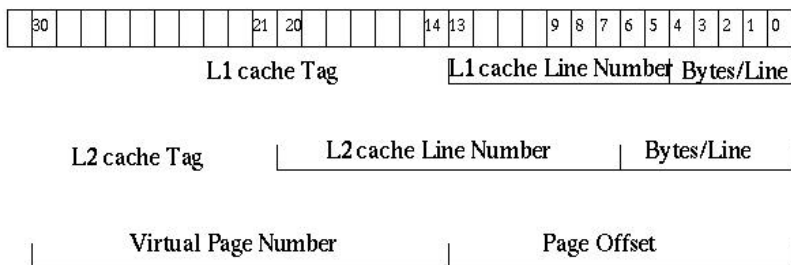
Cache Type	Total Size	Cache Line Size	# of cache lines	Mapping Method	Replacement Policy
L1 Instruction Cache	32KB	64-byte	512	2-way set associative	Least Recently Used (LRU)
L1 Data Cache	32KB	32-byte	1024	2-way set associative	LRU
L2 Unified Cache	4MB ; 8MB (for lomax)	128-byte	2**15=32768 ; 2**16=65536	2-way set associative	LRU

The L1 and L2 caches are divided into cache lines. For the L1 instruction cache, L1 data cache and L2 cache, the size of a cache line is 64 bytes, 32 bytes and 128 bytes, respectively. Thus, the number of cache lines for each cache is 32KB/64bytes= 512 lines for L1 instruction cache, 32KB/32bytes = 1024 lines for L1 data cache and 4MB/128 bytes= 2**15 lines for L2 cache (for lomax, there are 8MB/128bytes = 2**16 lines).

The use of cache line makes sure that a high bandwidth between the caches and memory is maintained. When a particular word is copied into cache, several nearby words are copied with it. If 1 word is 4-bytes long, one L1 data cache line can hold 8 consecutive data, and one L2 cache line can hold 32 consecutive data.

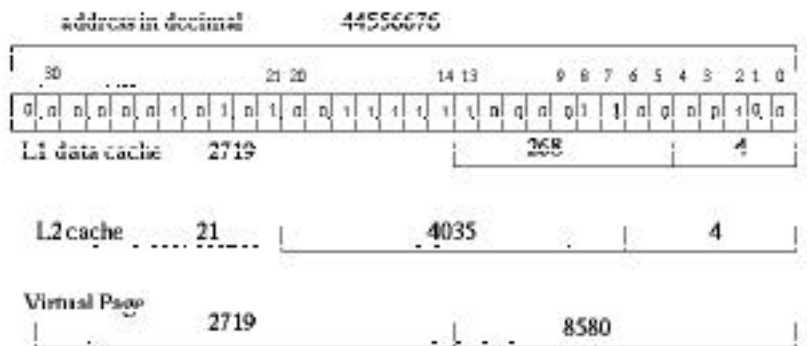
Two-Way Set Associativity

When a datum is loaded into either L1 or L2 cache, the cache line it goes to is not random. In fact, with the use of two-way set associative mapping method, there are only two specific cache lines that a datum can reside. The two cache lines that this datum can reside are determined by the middle bits of the address of this datum. In the case where the L1 cache size is 32KB with 32B per cache line and L2 cache size is 4MB with 128B per cache line, the following figure illustrates how many and which middle bits of the address are used to determine the cache lines a datum will use. An example is provided to further demonstrate how to determine the two L1 cache lines and two L2 cache lines a datum can use.



Example : Determination of the Cache Lines for a Datum

For ease of discussion, all numbers will be converted from binary to decimal in this example.



If the base address of a datum is represented by 44556676 in decimal, the two L2 cache lines this datum can reside are line number 4035 and 8131 as determined below:

$44556676 / (4\text{MB} / 2 \text{ way}) = 21 \text{ remainder } 516484$

$516484 / 128\text{B} \text{ (L2 cache line size)} = 4035 \text{ remainder } 4$

The first L2 cache line this datum can reside = 4035

The second L2 cache line this datum can reside = $4035 + (2 * 15 \text{ lines} / 2 \text{ way}) = 8131$

Similarly, for the two L1 data cache lines,

$44556676 / (32\text{KB} / 2 \text{ way}) = 2719 \text{ remainder } 8580$

$8580 / 32\text{B} \text{ (L1 data cache line size)} = 268 \text{ remainder } 4$

The first L1 data cache line this data can reside = 268

The second L1 data cache line this data can reside = $268 + (1024 \text{ lines} / 2 \text{ way}) = 780$

To determine if a datum is in cache, one only needs to check these two specific cache lines. Once the two cache lines are determined, one checks the cache tag on each of the two lines to see if it matches the rest of the address. If it does, this cache line contains our data and we have a cache hit. If the cache tag does not match the rest of our address, we have a cache miss and the data must be loaded from the next level of the memory hierarchy.

If the program refers to two data whose middle address bits are identical, copies of both can be maintained in the cache simultaneously. When a third datum with the same middle address bits is accessed, it replaces one of the two previous data - the one that was least recently accessed.

Example : L2 Cache Trashing among 3 Data with Identical Middle Address Bits

In the following program, each one of the 3 arrays a, b and c occupy exactly 2MB of memory and thus $a(i,j)$, $b(i,j)$ and $c(i,j)$ have identical low-21 bits (bits 20:0) in their base addresses. Lets also assume that the base address of $a(1,1)$ is 44556676 in decimal. Thus $a(1,1)$ can only reside in L2 cache line numbers 4035 or 8131. If $a(1,1)$ is in line 4035, then $b(1,1)$ can reside in line number 8131. When $c(1,1)$ is needed, an L2 cache line that contains either $a(1,1)$ or $b(1,1)$ needs to be replaced with data containing $c(1,1)$ because $c(1,1)$ can only reside in either line number 4035 or 8131 as $a(1,1)$ and $b(1,1)$.

```
dimension /abc/ a(1024,512), b(1024,512), c(1024,512)
do j=1,512
do i=1,1024
a(i,j)=a(i,j)+b(i,j)+c(i,j)
end do
end do
```

Quiz : If the above code is run in systems (as in Iomax) whose L2 cache is two-way set associative, its size is 8MB and each L2 cache line is 128B, will the two L2 cache lines for $a(1,1)$ be the same as those two lines for $b(1,1)$? What about $c(1,1)$?

The Situations that Cache Latency Applies

A cache miss occurs when the desired data is not found. A delay (latency) applies when:

- The cache line(s) where the desired data should reside is empty.
- The cache line needs to flush its current contents (which have been modified) back to memory, and replaced with the desired data.
- The contents of the cache line have not been modified and will be discarded in order to load in the desired data.

Principles of Good Cache Use

- A program ought to make use of every word of every cache line that it touches.

Clearly, if a program does not make use of every word in a cache line, the time spent loading the unused parts of the line is wasted.

- A program should use a cache line intensively and then not return to it later.

When a program visits a cache line a second time after a delay, the cache line may have been displaced by other data, so the program is delayed twice waiting for the same data.

Memory Access Latency

Latency of memory access is best measured in CPU clock cycles. For a 250MHz CPU, 1 clock cycle is 4 nanoseconds. The latency of data access becomes greater with each cache level. A miss at each level of the memory hierarchy multiplies the latency by an order of magnitude or more. Clearly a program can sustain high performance only by achieving a very high ratio of cache hits at every level.

Memory Hierarchy	Latency (in clock periods)
CPU Register	0
L1 cache hit	2/3
L1 cache miss satisfied by L2 cache hit	8/10
L2 cache miss satisfied from main memory, no TLB miss	75/250 (depending on the node where the memory resides)
TLB miss requiring only reload of the TLB to refer to a virtual page already in memory (minor page fault)	~ 2000
TLB miss requiring virtual page to load from disk (major page fault)	~ hundreds of millions

PERFORMANCE PROFILING

- Do your performance profiling using dedicated node(s)
Measurement of the performance of your code on a shared environment (for example, turing) is misleading. Do your measurement using dedicated nodes by submitting it through PBS in order to get nodes dedicated to your job.
- Remember that performance could still vary even with dedicated node(s)
The fact that the memory location your job use (if more than 1 node for hopper and steger, and more than 8 nodes on lomax) may be different from run to run means that you may get different performance.
- Measure change in performace only on the same machine

Available Tools

- **time, timex, ssusage, etime, dtime, second, timef**
 - **time**
time - return elapsed time, user time and system time in seconds to standard error.
time may be directed to produce other resource usage reports via command line options. Read man page for details.
Example : /bin/time a.out

```
real 1.415
user 1.350
sys 0.043
```
 - **timex**
timex - return elapsed time, user time and system time in seconds. It also reports process data and system activity when options are used. Read man page for details.
I found that the options are only working on turing, not hopper and lomax
 - **ssusage**
ssusage - return elapsed time, user time and system time and other resources used. ssusage is a SpeedShop executable. Read man page for details.
Example : /bin/ssuage a.out
The usage information is printed to stderr in the form:
1.42 real, 1.35 user, 0.04 sys, 0 majf, 39 minf, 0 sw, 1 rb, 0 wb, 37 vcx, 0 icx, 5696 mxrss
 - **etime, dtime, second functions**
etime - return user+system execution time in seconds since the start of execution of the program.
dtime - return user+system execution time in seconds since the last call to dtime, or the start of execution on the first call.
second - returns the user time for a process in seconds since the start of execution of the program. The implementation on SGI of second gets the time from the system function ETIME.
Note : etime, dtime and second functions have to be declared as real*4 even if the rest of program use real*8.
- **Example : Time profiling using etime**
Sample program : etime.f

```

program etime
! etime is called before and after sub is called
! To find the time spent in sub, one has to
! calculate the difference between t1 and t2

real*8 a(1024,1024)
real*4 etime, tarray(2)

pi= 3.14159

call random_number(a)

do k=1,10
  p1 = pi**(k)

  t1=etime(tarray)
  write (6,*) 't1 = ', t1
  call sub(a,pi)
  t2=etime(tarray)
  write (6,*) 't2 = ', t2

end do

stop
end

subroutine sub(a,b)
real*8 a(1024,1024)

do j=1,1024
  do i=1,1024
    a(i,j)=a(i,j)/b
  end do
end do

return
end

```

To compile and execute

```

f90 -o etimerun -64 -r8 etime.f
etimerun > etime.out &

```

The output

```

t1 = 0.25039100646972656
t2 = 0.53308498859405518
t1 = 0.53329503536224365
t2 = 0.79606801271438599
t1 = 0.79623997211456299
t2 = 1.0392590761184692
t1 = 1.0394420623779297
t2 = 1.2822780609130859
t1 = 1.2824519872665405
t2 = 1.5241659879684448
t1 = 1.5243419408798218
t2 = 1.7203249931335449
t1 = 1.7204980850219727
t2 = 1.9164799451828003
t1 = 1.9166569709777832
t2 = 2.112468957901001
t1 = 2.1126461029052734
t2 = 2.3085849285125732
t1 = 2.3087630271911621
t2 = 2.5047061443328857 # Notice the values for t1 and t2 are increasing.

```

Example : Time profiling using dtime

Sample program : dtime.f

```

program dtime
! dtime is called before and after sub is called
! t2 gives the time spent in sub.

real*8 a(1024,1024)
real*4 dtime, tarray(2)

pi= 3.14159

call random_number(a)

do k=1,10
  p1 = pi**(k)

  t1=dtime(tarray)
  write (6,*) 't1 = ', t1
  call sub(a,pi)
  t2=dtime(tarray)
  write (6,*) 't2 = ', t2

end do

stop
end

subroutine sub(a,b)
real*8 a(1024,1024)

do j=1,1024
  do i=1,1024
    a(i,j)=a(i,j)/b
  end do
end do

return
end

```

To compile and execute

```

f90 -o dtimerun -64 -r8 dtime.f
dtimerun > dtime.out &

```

The output

```
t1 = 0.24668100476264954
t2 = 0.29617798328399658
t1 = 1.42000004416331649E-4
t2 = 0.25391101837158203
t1 = 1.47999991895630956E-4
t2 = 0.25369700789451599
t1 = 1.44999998155981302E-4
t2 = 0.24676500260829926
t1 = 1.49999992572702467E-4
t2 = 0.25284498929977417
t1 = 1.50000007124617696E-4
t2 = 0.20771600306034088
t1 = 1.42000004416331649E-4
t2 = 0.20047000050544739
t1 = 1.50999985635280609E-4
t2 = 0.20701700440989868
t1 = 1.53000000864267349E-4
t2 = 0.20787800848484039
t1 = 1.47999991895630956E-4
t2 = 0.20037999749183655 #Notice that the value for every t2 is about 0.2 sec
```

o **timef function**

- timef - return elapsed wall-clock time in **milliseconds** since the previous call to timef
- timef function has to be declared as real*8 or real and compiled with -r8 option
 - Current default compiler version MIPSpro.7.3.1.1m does not work with timef. The Fortran libraries libfortran.so and libffio.so contain subroutines with the same name. This can cause timef() to return NaN or Infinity.
 - Use older versions of compiler such as MIPSpro.7.2.1.1m, MIPSpro.7.2.1.2m, MIPSpro.7.3.0.0 or the new version **MIPSpro.7.3.1.2m** if you want to use timef.

● **Perfex (Hardware Performance Counter)**

The R10000 and R12000 processors include counters that can be used to count the frequency of events during the execution of a program. Such frequency can be translated into an estimated time spent for each event and thus will be useful in understanding the nature of the code and in optimizing it.

The R10000 processor supplies two performance counters for counting certain hardware events. Each counter can track one event at a time and there are a choice of sixteen events per counter. The first counter track events 0 - 15 and the second one tracts events 16-31. There are also two associated control registers which are used to specify which event the relevant counter is counting.

The R12000 processor supplies two performance counters for counting hardware events. Each counter can track one event at a time, and you can choose among 32 events per counter.

o **The 32 R10000 Events**

On systems with **R10000** processors, these events are as follows:

Event	Description
0	Cycles
1	Issued instructions
2	Issued loads
3	Issued stores
4	Issued store conditionals
5	Failed store conditionals
6	Decoded branches or Resolved branches (depending on versions of R10000)
7	Quadwords written back from secondary cache
8	Correctable secondary cache data array ECC errors
9	Primary instruction cache misses
10	Secondary instruction cache misses
11	Instruction misprediction from secondary cache way prediction table
12	External interventions
13	External invalidations
14	Virtual coherency conditionals or ALU/FPU forward progress cycles (depending on versions of R10000)
15	Graduated instructions
16	Cycles
17	Graduated instructions
18	Graduated loads
19	Graduated stores
20	Graduated store conditionals
21	Graduated floating-point instructions
22	Quadwords written back from primary data cache
23	Translation lookaside buffer (TLB misses)
24	Mispredicted branches
25	Primary data cache misses
26	Secondary data cache misses

27	Data misprediction from secondary cache way prediction table
28	External intervention hits in secondary cache
29	External invalidation hits in secondary cache
30	Store/prefetch exclusive to clean block in secondary cache
31	Store/prefetch exclusive to shared block in secondary cache

Detail description of the R10000 Counter Event can be found in [Appendix B](#) of SGI's [Origin 2000 and Onyx Performance Tuning and Optimization Guide](#).

o **The 32 R12000 Events**

On systems with **R12000** processors, these events are as follows:

Event	Description
0	Cycles
1	Issued instructions or Decoded instructions
2	Decoded loads
3	Decoded stores
4	Miss handling table occupancy
5	Failed store conditionals
6	Resolved conditional branches
7	Quadwords written back from secondary cache
8	Correctable secondary cache data array ECC errors
9	Primary instruction cache misses
10	Secondary instruction cache misses
11	Instruction misprediction from secondary cache way prediction table
12	External interventions
13	External invalidations
14	Not implemented
15	Graduated instructions
16	Executed prefetch instructions
17	Prefetch primary data cache misses
18	Graduated loads
19	Graduated stores
20	Graduated store conditionals
21	Graduated floating-point instructions
22	Quadwords written back from primary data cache
23	Translation lookaside buffer (TLB misses)
24	Mispredicted branches
25	Primary data cache misses
26	Secondary data cache misses
27	Data misprediction from secondary cache way prediction table
28	State of intervention hits in secondary cache
29	State of invalidation hits in secondary cache
30	Store/prefetch exclusive to clean block in secondary cache
31	Store/prefetch exclusive to shared block in secondary cache

o **perfex command**

The perfex commands provide convenient interfaces to hardware counter information. Use this command to measure the **performance of the entire program**.

- ☐ **%perfex [options] executable_name [arguments]**
- ☐ options:

-e (cnt1) -e (cnt2)	specific counts
-a	all counts
-x	exceptional level counting
-y	time estimates, metrics
-mp	all data per thread
-t	table of ideal numbers

Note : On R10000 processors (hopper and steger), the default event to be tracked by the first counter is event 0 and that by the second counter is event 16. If two events that belong to the same counter are chosen by the user, one of them will be automatically changed to the default event. For example, "-e 23 -e 25" will be changed to "-e 0 -e 25"; "-e 7 -e 12" will be changed to "-e 7 -e 16". This seems to also apply to R12000 processors that lomax uses.

Option -a produces counts for all events by multiplexing over 16 events per counter. The OS does the switching round robin at clock interrupt boundaries. The resulting counts are normalized by multiplying by 16 to give an estimate of the values they would have had for exclusive counting. Due to the equal-time nature of the multiplexing, events present in large enough numbers to contribute significantly to the execution time will be fairly represented. Events concentrated in a few short regions (for instance, instruction cache misses) may not be projected very accurately.

- You need to set the hardware counters to user mode (default is at global mode) when you submit your job to PBS. Specifically, you can do this (by requesting hpm=1) on the command line

```
%qsub -l hpm=1 job_script
```

or include the following in your PBS script

```
#PBS -l hpm=1
```

Example : To find the cost (in cycles) for 1 occurrence of each event on the 250MHz R10000 hopper and steger

Sample PBS script

```
#PBS -l hpm=1
/bin/time perfex -a -t #no executable needed for option -t
```

The perfex output is sent to standard error

```
/bin/time perfex -a -t
WARNING: Multiplexing events to project totals--inaccuracy possible.

Costs for IP27 processor
MIPS R10000 CPU
CPU revision 3.x
```

Event Counter Name	Typical Cost	Minimum Cost	Maximum Cost	
0 Cycles.....	1.00 clks	1.00 clks	1.00 clks	
1 Issued instructions.....	0.00 clks	0.00 clks	1.00 clks	
2 Issued loads.....	1.00 clks	1.00 clks	1.00 clks	
3 Issued stores.....	1.00 clks	1.00 clks	1.00 clks	
4 Issued store conditionals.....	1.00 clks	1.00 clks	1.00 clks	
5 Failed store conditionals.....	1.00 clks	1.00 clks	1.00 clks	
6 Decoded branches.....	1.00 clks	1.00 clks	1.00 clks	
7 Quadwords written back from scache.....	6.40 clks	4.23 clks	6.40 clks	
8 Correctable scache data array ECC errors.....	0.00 clks	0.00 clks	1.00 clks	
9 Primary instruction cache misses.....	18.02 clks	5.63 clks	18.02 clks	
10 Secondary instruction cache misses.....		75.50 clks	49.36 clks	84.00 clks
11 Instruction misprediction from scache way prediction table.....	0.00 clks	0.00 clks	1.00 clks	
12 External interventions.....	0.00 clks	0.00 clks	0.00 clks	
13 External invalidations.....	0.00 clks	0.00 clks	0.00 clks	
14 ALU/FPU progress cycles.....	1.00 clks	1.00 clks	1.00 clks	
15 Graduated instructions.....	0.00 clks	0.00 clks	1.00 clks	
16 Cycles.....	1.00 clks	1.00 clks	1.00 clks	
17 Graduated instructions.....	0.00 clks	0.00 clks	1.00 clks	
18 Graduated loads.....	1.00 clks	1.00 clks	1.00 clks	
19 Graduated stores.....	1.00 clks	1.00 clks	1.00 clks	
20 Graduated store conditionals.....	1.00 clks	1.00 clks	1.00 clks	
21 Graduated floating point instructions.....	1.00 clks	0.50 clks	52.00 clks	
22 Quadwords written back from primary data cache.....	3.85 clks	3.14 clks	4.45 clks	
23 TLB misses.....		68.09 clks	68.09 clks	68.09 clks
24 Mispredicted branches.....		1.42 clks	0.64 clks	5.22 clks
25 Primary data cache misses.....	9.01 clks	2.82 clks	9.01 clks	
26 Secondary data cache misses.....		75.50 clks	49.36 clks	84.00 clks
27 Data misprediction from scache way prediction table.....	0.00 clks	0.00 clks	1.00 clks	
28 External intervention hits in scache.....	0.00 clks	0.00 clks	0.00 clks	
29 External invalidation hits in scache.....	0.00 clks	0.00 clks	0.00 clks	
30 Store/prefetch exclusive to clean block in scache.....	1.00 clks	1.00 clks	1.00 clks	
31 Store/prefetch exclusive to shared block in scache.....	1.00 clks	1.00 clks	1.00 clks	
real 0.298				
user 0.026				
sys 0.123				

Example : To find the cost (in cycles) for 1 occurrence of each event on the 400MHz R12000 lomax

The perfex output

WARNING: Multiplexing events to project totals--inaccuracy possible

Costs for IP27 processor MIPS R12000 CPU				
Event Counter Name	Typical Cost	Minimum Cost	Maximum Cost	
0 Cycles.....	1.00 clks	1.00 clks	1.00 clks	
1 Decoded instructions.....	0.00 clks	0.00 clks	1.00 clks	
2 Decoded loads.....	1.00 clks	1.00 clks	1.00 clks	
3 Decoded stores.....	1.00 clks	1.00 clks	1.00 clks	
4 Miss handling table occupancy.....	1.00 clks	1.00 clks	1.00 clks	
5 Failed store conditionals.....	1.00 clks	1.00 clks	1.00 clks	
6 Resolved conditional branches.....	1.00 clks	1.00 clks	1.00 clks	
7 Quadwords written back from scache.....	8.49 clks	5.90 clks	8.77 clks	
8 Correctable scache data array ECC errors.....	0.00 clks	0.00 clks	1.00 clks	
9 Primary instruction cache misses.....	17.01 clks	4.34 clks	17.01 clks	
10 Secondary instruction cache misses.....		99.89 clks	63.03 clks	99.89 clks
11 Instruction misprediction from scache way prediction table.....	0.00 clks	0.00 clks	1.00 clks	
12 External interventions.....	0.00 clks	0.00 clks	0.00 clks	
13 External invalidations.....	0.00 clks	0.00 clks	0.00 clks	
14 ALU/FPU progress cycles.....	1.00 clks	1.00 clks	1.00 clks	
15 Graduated instructions.....	0.00 clks	0.00 clks	1.00 clks	
16 Executed prefetch instructions.....	0.00 clks	0.00 clks	0.00 clks	
17 Prefetch primary data cache misses.....	0.00 clks	0.00 clks	1.00 clks	
18 Graduated loads.....	1.00 clks	1.00 clks	1.00 clks	
19 Graduated stores.....	1.00 clks	1.00 clks	1.00 clks	
20 Graduated store conditionals.....	1.00 clks	1.00 clks	1.00 clks	
21 Graduated floating point instructions.....	1.00 clks	0.50 clks	52.00 clks	
22 Quadwords written back from primary data cache.....	3.98 clks	3.14 clks	3.98 clks	
23 TLB misses.....		77.78 clks	77.78 clks	77.78 clks
24 Mispredicted branches.....		7.28 clks	6.00 clks	8.81 clks
25 Primary data cache misses.....	8.50 clks	2.17 clks	8.50 clks	
26 Secondary data cache misses.....		99.89 clks	63.03 clks	99.89 clks
27 Data misprediction from scache way prediction table.....	0.00 clks	0.00 clks	1.00 clks	
28 State of intervention hits in scache.....	0.00 clks	0.00 clks	0.00 clks	
29 State of invalidation hits in scache.....	0.00 clks	0.00 clks	0.00 clks	
30 Store/prefetch exclusive to clean block in scache.....	1.00 clks	1.00 clks	1.00 clks	
31 Store/prefetch exclusive to shared block in scache.....	1.00 clks	1.00 clks	1.00 clks	

Quiz : Compare the outputs from Example 1 and Example 2. Why does it cost more cycles on the 400MHz R12000 machine than on the 250MHz R10000 machine for events such as 10 (Secondary instruction cache misses), 23 (TLB misses), 24 (Mispredicted branches) and 26 (Secondary data cache misses) ?

Example : Use `perfx -a -x -y` to find the cause of bad performance

In this example, `perfx` is used to identify the cause of bad performance in a program. Secondary data cache miss is found to be the main cause in this program as demonstrated in the `perfx` output.

Sample program : `L2_cache_trash.f`

```

1  program L2_cache_trashing
2
3! Arrays a, b, c, and d are all 4MB in size. Accessing a(i,j), b(i,j)
4! c(i,j), and d(i,j) simultaneously causes L2_cache_trashing when
5! the size of L2 cache is either 4MB (hopper, steger) or 8MB (lomax)
6! with 2-way set associativity
7
8  dimension a(1024,1024), b(1024,1024), c(1024,1024), d(1024,1024)
9
10 call random_number(b)
11 call random_number(c)
12 call random_number(d)
13
14 do j=1,1024
15 do i=1,1024
16 a(i,j)=b(i,j)+c(i,j)*d(i,j)
17 end do
18 end do
19
20 write (12) a
21
22 stop
23 end

```

To compile and execute on hopper

`%f90 -o L2_cache_trash L2_cache_trash.f ! default optimization level is -O0`

Performance output from "`/bin/time perfx -a -x -y L2_cache_trash`"

```

/bin/time perfex -a -x -y L2_cache_trash
WARNING: Multiplexing events to project totals--inaccuracy possible.
Summary for execution of L2_cache_trash

Based on 250 MHz IP27
MIPS R10000 CPU
CPU revision 3.x

Event Counter Name          Counter Value   Typical
                               Time (sec)         Time (sec)
                               Minimum           Maximum
                               Time (sec)         Time (sec)
=====
0 Cycles.....              653654512      2.614618      2.614618      2.614618
16 Cycles.....              653654512      2.614618      2.614618      2.614618
26 Secondary data cache misses.....          4178336      1.261857      0.824971      1.403921
14 ALU/FPU progress cycles.....            82554064      0.330216      0.330216      0.330216
7 Quadwords written back from scache.....        8635504      0.221069      0.146113      0.221069
25 Primary data cache misses.....           4348624      0.156724      0.049052      0.156724
2 Issued loads.....          29675728      0.118703      0.118703      0.118703
18 Graduated loads.....          29524944      0.118100      0.118100      0.118100
3 Issued stores.....          11268688      0.045075      0.045075      0.045075
19 Graduated stores.....          11248768      0.044995      0.044995      0.044995
22 Quadwords written back from primary data cache.....        2475040      0.038116      0.031087      0.044056
21 Graduated floating point instructions.....       7787744      0.031151      0.015575      0.1619851
6 Decoded branches.....          4437184      0.017749      0.017749      0.017749
10 Secondary instruction cache misses.....         7664      0.002315      0.001513      0.002575
9 Primary instruction cache misses.....         11936      0.000860      0.000269      0.000860
23 TLB misses.....           1200      0.000327      0.000327      0.000327
24 Mispredicted branches.....           7872      0.000045      0.000020      0.000164
4 Issued store conditionals.....            80      0.000000      0.000000      0.000000
30 Store/prefetch exclusive to clean block in scache.....         64      0.000000      0.000000      0.000000
20 Graduated store conditionals.....          16      0.000000      0.000000      0.000000
31 Store/prefetch exclusive to shared block in scache.....          16      0.000000      0.000000      0.000000
1 Issued instructions.....          122630720      0.000000      0.000000      0.000000
5 Failed store conditionals.....            0      0.000000      0.000000      0.000000
8 Correctable scache data array ECC errors.....            0      0.000000      0.000000      0.000000
11 Instruction misprediction from scache way prediction table..         576      0.000000      0.000000      0.000002
12 External interventions.....          2144      0.000000      0.000000      0.000000
13 External invalidations.....          9776      0.000000      0.000000      0.000000
15 Graduated instructions.....        167980368      0.000000      0.000000      0.671921
17 Graduated instructions.....        134798896      0.000000      0.000000      0.539196
27 Data misprediction from scache way prediction table.....         1968      0.000000      0.000000      0.000008
28 External intervention hits in scache.....         1936      0.000000      0.000000      0.000000
29 External invalidation hits in scache.....         2240      0.000000      0.000000      0.000000

Statistics
=====
Graduated instructions/cycle.....          0.256986
Graduated floating point instructions/cycle.....        0.011914
Graduated loads & stores/cycle.....          0.062378
Graduated loads & stores/floating point instruction.....        5.235626
Mispredicted branches/Decoded branches.....        0.001774
Graduated loads/Issued loads.....          0.994919
Graduated stores/Issued stores.....          0.998232
Data mispredict/Data scache hits.....          0.011557
Instruction mispredict/Instruction scache hits.....        0.134831
L1 Cache Line Reuse.....          8.376233
L2 Cache Line Reuse.....          0.040755 This is terrible !!
L1 Data Cache Hit Rate.....          0.893347
L2 Data Cache Hit Rate.....          0.039159 This is terrible !!
Time accessing memory/Total time.....          0.605061
Time not making progress (probably waiting on memory) / Total time.....        0.873704
L1--L2 bandwidth used (MB/s, average per process).....        68.368153
Memory bandwidth used (MB/s, average per process).....       257.397088
MFLOPS (average per process).....          2.978540

real 3.552
user 2.705
sys 0.377

```

o libperfex

A few functions are provided in libperfex to allow simple access to the hardware event counters in your program. Use these functions if you want to measure the **performance of sections of your program rather than the entire program**.

C SYNOPSIS

```

int start_counters( int e0, int e1 );
int read_counters( int e0, long long *c0, int e1, long long *c1);
int print_counters( int e0, long long c0, int e1, long long c1);
int print_costs( int e0, long long c0, int e1, long long c1);
int load_costs(char *CostFileName);

```

FORTRAN SYNOPSIS

```

INTEGER*8 c0, c1
INTEGER e0, e1
CHARACTER(*p) CostFileName
INTEGER*4 function start_counters( e0, e1 )
INTEGER*4 function read_counters( e0, c0, e1, c1 )
INTEGER*4 function print_counters( e0, c0, e1, c1 )
INTEGER*4 function print_costs( e0, c0, e1, c1 )
INTEGER*4 function load_costs( CostFileName )

```

- e0 and e1 : int types specifying which events to count
- c0 and c1 : count values of the specified events
- start_counters : zero out the internal software counters and then start counters
- read_counters : read the counters and then stop them
- print_counters : print the counts of specified events to standard error
- print_costs : print the counts together with approximate time estimates as described for perfex.
- load_costs : load a cost table from a file. If /etc/perfex.costs is present, it is used instead.
- call start_counters immediately before section of code to be measured
- call read_counters and print_counters immediately after section of code to be measured
- if you want to accumulate counts over multiple start/read calls, you must save out the counts and do the accumulation yourself
- include libperfex.a in your link/load

f90 -o executable program.f -lperfex

- You need to set the hardware counters to user mode (default is at global mode) when you submit your job to PBS. Specifically, you can do this (by requesting hpm=1) on the command line

```
%qsub -l hpm=1 job_script
```

or include the following in your PBS script

```
#PBS -l hpm=1
```

- The use of libperfex is currently working on lomax, **not on hopper and steger**.

Example : Use libperfex to measure performance**Sample program : L2_cache_trash_libperfex.f**

```

      program L2_cache_trash_libperfex
! Use libperfex routines to measure the performance of various sections of the code
! In this example, event 0 (cycles) and 26 (L2 cache miss) are monitored
! Compare the counts from each section to find where the performance
! bottleneck is

      dimension a(1024,1024), b(1024,1024), c(1024,1024), d(1024,1024)

      integer*8 c0, c1, c0sum, c1sum
      integer    e0, e1

      e0 = 0
      e1 = 26

      call start_counters(e0,e1)

! Section 1
      call random_number(b)
      call random_number(c)
      call random_number(d)

      call read_counters(e0,c0,e1,c1)
      call print_counters(e0,c0,e1,c1)
      call print_costs(e0,c0,e1,c1)

      c0sum=c0
      c1sum=c1

      e0 = 0
      e1 = 26
! you can choose different events to monitor
! here I use the same event as in pervious
! section

      call start_counters(e0,e1)

! Section 2
      do j=1,1024
      do i=1,1024
      a(i,j)=b(i,j)+c(i,j)*d(i,j)
      end do
      end do

      call read_counters(e0,c0,e1,c1)
      call print_counters(e0,c0,e1,c1)
      call print_costs(e0,c0,e1,c1)

      c0sum=c0sum+c0
      c1sum=c1sum+c1

      e0 = 0
      e1 = 26

      call start_counters(e0,e1)

!Section 3
      write (12) a

      call read_counters(e0,c0,e1,c1)
      call print_counters(e0,c0,e1,c1)
      call print_costs(e0,c0,e1,c1)

      c0sum=c0sum+c0
      c1sum=c1sum+c1

      call print_costs(e0,c0sum,e1,c1sum)

      stop
      end

```

To compile and execute on lomax

%f90 -o L2_cache_trash_libperfex L2_cache_trash_libperfex.f -lperfex

libperfex output (standard error)

```

/bin/time L2_cache_trash_libperfex

! Section 1

Event Counter Name                      Counter Value
=====
0 Cycles.....                        81779561
26 Secondary data cache misses.....    587

Based on 400 MHz IP27
MIPS R12000 CPU
Typical Minimum Maximum
Event Counter Name Counter Value Time (sec) Time (sec) Time (sec)
=====
0 Cycles.....      81779561 0.204449 0.204449 0.204449
26 Secondary data cache misses.....    587 0.000147 0.000092 0.000147

! Section 2

Event Counter Name                      Counter Value
=====
0 Cycles.....                        779856591
26 Secondary data cache misses.....   4189221

Based on 400 MHz IP27
MIPS R12000 CPU
Typical Minimum Maximum
Event Counter Name Counter Value Time (sec) Time (sec) Time (sec)
=====
0 Cycles.....      779856591 1.949641 1.949641 1.949641
26 Secondary data cache misses.....   4189221 1.046153 0.660116 1.046153

! Section 3

Event Counter Name                      Counter Value
=====
0 Cycles.....                        568271
26 Secondary data cache misses.....     897

Based on 400 MHz IP27
MIPS R12000 CPU
Typical Minimum Maximum
Event Counter Name Counter Value Time (sec) Time (sec) Time (sec)
=====
0 Cycles.....      568271 0.001421 0.001421 0.001421
26 Secondary data cache misses.....     897 0.000224 0.000141 0.000224

! 3 Sections Combined

Based on 400 MHz IP27
MIPS R12000 CPU
Typical Minimum Maximum
Event Counter Name Counter Value Time (sec) Time (sec) Time (sec)
=====
0 Cycles.....      862204423 2.155511 2.155511 2.155511
26 Secondary data cache misses.....   4190705 1.046524 0.660350 1.046524

real 3.287
user 2.171
sys 0.782

```

• SpeedShop - ssrun, prof and pixie

SpeedShop is the generic name for an integrated package of performance tools to run performance experiments on executables, and to examine the results of those experiments. A [SpeedShop User's Guide](http://techpubs.sgi.com/library/dynaweb_bin/ebt-bin/0650/nph-infosrch.cgi/infosrchtpl/SGL_Developer/SShop_UG) (http://techpubs.sgi.com/library/dynaweb_bin/ebt-bin/0650/nph-infosrch.cgi/infosrchtpl/SGL_Developer/SShop_UG) is available in SGI's techpubs library if you want to master the use of SpeedShop.

- o SpeedShop provides the following profiling experiment types:
 - PC sampling
 - Ideal time
 - User time
 - Hardware counter profiling
 - Floating point exception tracing
 - Heap tracing
- o **ssrun** collects SpeedShop performance data
- o **pixie** sets SpeedShop in expert mode and measure the frequency of code execution
- o **prof** analyzes and displays performance data collected by ssrun and pixie
- o SpeedShop does not require special compilation or re-linking
- o SpeedShop.1.4 is required in order for the ideal time experiment type to work. On **hopper**, currently you need to include the following in your PBS script until SpeedShop.1.4 version becomes the default on hopper: (This is **not required for steger and lomax**)

module load SpeedShop.1.4

- o For experiment types that involve hardware performance counters, you need to set hardware counters to user mode when you submit your job through PBS. Specifically, you can do this (by requesting hpm=1) on the command line:

```
%qsub -l hpm=1 job_script
```

or include the following in your PBS script

```
#PBS -l hpm=1
```

- o For ssrun with experiment types that involve hardware performance (*_hwc and *_hwctime) counters, it is working currently only on lomax, **not on hopper and steger**.

o ssrun - SpeedShop Experiment

□ Command

SpeedShop experiments are recorded using the ssrun command, as follows:

```
ssrun [ssrun-options] -exp_type executable_name [executable_args]
```

□ Options of Experiment Types

The following experiment types are supported on all architectures:

Experiment Type	Description
usertime	Returns CPU time , the time your program is actually running plus the time the operating system is performing services for your program. The display generated by prof breaks the program time down into the time used by each function within the program. Uses statistical callstack profiling , based on CPU time, with a time sample interval of 30 milliseconds . Note: An o32 executable must explicitly link with -lexc for these experiments to work. Program execution may show significant slowdown compared to the original executable. The stack unwind code sometimes fails to completely unwind the stack; consequently, caller attribution cannot be done beyond the point of failure.
[f]pcsamp[x]	Returns the estimated actual CPU time for each source code line, machine code line, and function in your program. Uses statistical PC sampling, using 16-bit bins, based on user and system time, with a sample interval of 10 milliseconds . If the optional f prefix is specified, a sample interval of 1 millisecond will be used. If the optional x suffix is specified, a 32-bit bin size will be used.
ideal	Returns the best possible time of which the program is capable. Uses basic-block counting , done by instrumenting the executable.
fpe	Traces all floating-point exceptions.
heap	Traces malloc and free calls and also supports various options for debugging heap usage. Use cvperf(1) to display this information; it is not supported with prof(1).
io	Traces the following I/O system calls: read(2), readv(2), write(2), writev(2), open(2), close(2), dup(2), pipe(2), and creat(2).
mpi	Traces calls to various MPI routines and generates a file viewable in the cvperf(1) performance analyzer window. For a list of the routines that are traced, see the ssrun(1) man page.

On machines with hardware performance counters (R10000 and R12000 machines), the following additional types are supported:

Experiment Type	Description
[f]gi_hwc	Uses statistical PC sampling, based on overflows of the graduated-instruction counter (counter17), at an overflow interval of 32771. If the optional f prefix is used, the overflow interval will be 6553.
[f]cy_hwc	Uses statistical PC sampling, based on overflows of the cycle counter (counter 0), at an overflow interval of 16411. If the optional f prefix is used, the overflow interval will be 3779.
[f]ic_hwc	Uses statistical PC sampling, based on overflows of the primary instruction-cache miss counter (counter 9), at an overflow interval of 2053. If the optional f prefix is used, the overflow interval will be 419.
[f]isc_hwc	Uses statistical PC sampling, based on overflows of the secondary instruction-cache miss counter (counter 10), at an overflow interval of 131. If the optional f prefix is used, the overflow interval will be 29.
[f]dc_hwc	Uses statistical PC sampling, based on overflows of the primary data-cache miss counter (counter 25), at an overflow interval of 2053. If the optional f prefix is used, the overflow interval will be 419.
[f]dsc_hwc	Uses statistical PC sampling, based on overflows of the secondary data-cache miss counter (counter 26), at an overflow interval of 131. If the optional f prefix is used, the overflow interval will be 29.
[f]tlb_hwc	Uses statistical PC sampling, based on overflows of the TLB miss counter (counter 23), at an overflow interval of 257. If the optional f prefix is used, the overflow interval will be 53.
[f]gfp_hwc	Uses statistical PC sampling, based on overflows of the graduated floating-point instruction counter (counter 21), at an overflow interval of 32771. If the optional f prefix is used, the overflow interval will be 6553.
[f]fsc_hwc	Uses statistical PC sampling, based on overflows of the failed store conditionals counter (counter 5), at an overflow interval of 2003. If the optional f prefix is used, the overflow interval will be 401.
prof_hwc	Uses statistical PC sampling, based on overflows of the counter specified by the environment variable <code>_SPEEDSHOP_HWC_COUNTER_NUMBER</code> , at an interval given by the environment variable <code>_SPEEDSHOP_HWC_COUNTER_OVERFLOW</code> . Note that these environment variables cannot be used to override the counter number or interval for the other defined experiments. They are examined only when the <code>prof_hwc</code> experiment is specified. The default counter is the primary instruction-cache miss counter and the default overflow interval is 2053.
gi_hwtime	Profiles the cycle counter using statistical call-stack sampling, based on overflows of the graduated- instruction counter (counter 17), at an overflow interval of 1000003.
cy_hwtime	Profiles the cycle counter using statistical call-stack sampling, based on overflows of the cycle counter (counter 16), at an overflow interval of 10000019.
ic_hwtime	Profiles the cycle counter using statistical call-stack sampling, based on overflows of the primary instruction-cache-miss counter (counter 9), at an overflow interval of 8009.
isc_hwtime	Profiles the cycle counter using statistical call-stack sampling, based on overflows of the secondary instruction-cache-miss counter (counter 10), at an overflow interval of 2003.
dc_hwtime	Profiles the cycle counter using statistical call-stack sampling, based on overflows of the primary data- cache-miss counter (counter 25), at an overflow interval of 8009.
dsc_hwtime	Profiles the cycle counter using statistical call-stack sampling, based on overflows of the secondary data- cache-miss counter (counter 26), at an overflow interval of 2003.
tlb_hwtime	Profiles the cycle counter using statistical call-stack sampling, based on overflows of the TLB miss counter (counter 23), at an overflow interval of 2521.
gfp_hwtime	Profiles the cycle counter using statistical call-stack sampling, based on overflows of the graduated floating-point instruction counter (counter 21), at an overflow interval of 10007.
fsc_hwtime	Profiles the cycle counter using statistical call-stack sampling, based on overflows of the failed store conditionals counter (counter 5), at an overflow interval of 5003.
prof_hwtime	Profiles the counter specified by the environment variable <code>_SPEEDSHOP_HWC_COUNTER_PROF_NUMBER</code> using statistical call-stack sampling, based on overflows of the counter specified by the environment variable <code>_SPEEDSHOP_HWC_COUNTER_NUMBER</code> , at an interval given by the environment variable <code>_SPEEDSHOP_HWC_COUNTER_OVERFLOW</code> . Note that these environment variables can not be used to override the counter numbers or interval for the other defined experiments. They are examined only when the <code>prof_hwtime</code> experiment is specified. The default overflow and profiling counter is the cycle counter and the default overflow interval is 10000019.

Example : PBS script for ssrun with usertime exp_type

```
#PBS -l walltime=0:10:00

cd /cluster/hopper/scratch1/userid <--- your working directory
ssrun -usertime L2_cache_trash
```

Example : PBS script for ssrun with ideal exp_type to be used on hopper

```
#PBS -l walltime=0:10:00

module load SpeedShop.1.4 ! not needed if submit to steger or lomax

cd /cluster/hopper/scratch1/userid
ssrun -ideal L2_cache_trash
```

Example : PBS script for ssrun with dsc_hwc exp_type

```
#PBS -l walltime=0:10:00
#PBS -l hpm=1

cd /cluster/hopper/scratch1/userid
ssrun -dsc_hwc L2_cache_trash    ! currently working only on lomax
```

□ Output Files

The result of an experiment is one or more files that are named by the following convention:

executable_name.exp_type.id

where id is one of the following one or two-letter codes followed by the process identifier (PID):

Letter Code	Description
m	For the master process created by <code>ssrun</code> ;
p	For a process created by a call to <code>sproc()</code> ;
f	For a process created by a call to <code>fork()</code> ;
e	For a process created by a call to <code>exec()</code> ;
s	For a process created by a call to <code>system()</code> ;
fe	For the <code>exec'd</code> process created by calls to <code>fork()</code> and <code>exec()</code> , with environment variable <code>_SPEEDSHOP_TRACE_FORK_TO_EXEC</code> set to <code>False</code> .

Example : output generated from `ssrun`

L2_cache_trash.usertime.m754877

L2_cache_trash.fpcsamp.m755026

L2_cache_trash.ideal.m1838173

L2_cache_trash.dsc_hwc.m1225967

o prof - SpeedShop Report Generation

□ Command

Report generation is done through the **prof** command:

prof [prof-options] executable_name.exp_type.id

Multiple files can be included only if they are recorded from the same executable with the same experiment type:

prof [prof-options] executable_name.exp_type.id1 executable_name.exp_type.id2

□ Output Options

Options to the `prof` command are grouped as follows:

- General options
- Output controls options
- Selectivity options
- CPU options
- Debugging options

Some of these options are described here. See `prof` man page to find details about the rest of the options.

```
-basicblocks    Prints a list of all the basic blocks executed, ordered
                by the total number of cycles spent in each basic
                block.  Works only with ideal experiments.

-b[butterfly]   Causes prof to print a report showing the callers and
                callees of each function, with inclusive time
                attributed to each.  For ideal experiments, the
                attribution is based on a heuristic.  For the various
call stack sampling and tracing experiments, the
                attribution is precise.  The usertime, totaltime, and
                some_hwctime experiments are statistical in nature and
                so are less exact.  This option is ignored for
                experiments in which the data does not support
                inclusive calculations.

-calipers [n1,n2] Causes prof to compute the data between caliper points
                n1 and n2, rather than for the entire experiment.  If n1
                >= n2, an error is reported.  If n1 is negative, it is
                set to the beginning of the experiment.  If n2 is
                greater than the maximum recorded, it is set to the
                maximum.  If n1 is omitted, zero is assumed.

-h[heavy]       Reports the most heavily used lines in descending order
                of use.  This option can be used when generating
                reports for ideal, pcsamp, or _hwc experiments.  It is
                ignored for other experiments.

-l[lines]       Performs like -h[heavy], but groups lines by procedure,
                with procedures sorted in descending order of use.
                Within a procedure, lines are listed in source order.
                This option can be used when generating reports for
ideal, pcsamp, or _hwc experiments.  It is ignored for
                other experiments.

-u[sage]        Prints a report on system statistics.

-dsolist        List all the DSO's in the program and their start and end text
                addresses.

-calls          Sorts function list by procedure calls rather than by
                time.  This option can only be used when generating
                reports for ideal experiments or for basic block
                counting data obtained with pixie.

-nh            Suppress various header blocks from the output.

-q[uit] n or n% or ncum% Truncates the -h[heavy], -l[lines], and -b[butterfly]
                listings after the first n lines have been listed,
                after those lines up to the one which takes more than n
                percent of the total, or after those lines up to the
                one that brings the cumulative total to more than n
                percent.  For example, -q 15 truncates each part of the
                listing after 15 lines of text, -q 15% truncates each
                part after the first line that represents less than 15
                percent of the whole, and -q 15cum% truncates each part
                after the line that brought the cumulative percentage
                above 15 percent.  For -b[butterfly], -q ncum% behaves
                the same as -q n%.

-clock megahertz Sets the CPU clock speed to megahertz MHz.  It alters
                the appropriate parts of the listing to reflect the
                clock speed.  The default value is the clock speed of
                the machine on which the experiment is performed.
```

□ Output Format

- prof writes an analysis of the performance data to stdout
- a summary of the experiment and description of the environment in which it was recorded
- a header that summarizes the particular data recorded
- function list

For all experiments, it produces a list of functions, annotated with the appropriate metric.

Experiment Type	Function List Annotation
usertime	exclusive time
totaltime	exclusive time
[f]pcsamp[x]	exclusive time
ideal	exclusive ideal time
_hwc experiments	exclusive counts
_hwc time experiments	exclusive time
fpe	exclusive FPEs
io	exclusive IO calls

- butterfly function list

If the -b[butterfly] flag is added, a list of callers and callees of each function is also produced.

- line list

If the -h[eavy] or -l[ines] options are included for pcsamp, _hwc, and ideal experiments, a report of data at the source line level is appended. The report is sorted either by the performance metric computed on a line basis, or by functions and then by line numbers within a function. For other experiments, these options are ignored.

- basicblock list

For ideal experiments only, if -basicblocks is specified, a report of data at the basic-block level is appended. If -archinfo is specified, also for ideal experiments only, a summary report of register usage, instruction usage, and various other statistics is appended. For other experiments, these options are ignored.

- dso list

If -dsolist is specified, a list of the DSOs used by the program is appended.

- resource usage data

If -u[sage] is specified, a summary of the resources used by the program is appended.

- Example : prof -b -q 8 -dsolist L2_cache_trash.usertime.m754877

```

SpeedShop profile listing generated Tue Jan 30 16:21:03 2001
prof -b -q 8 -dsolist L2_cache_trash.usertime.m754877
    L2_cache_trash (n32): Target program
        usertime: Experiment name
        ut:cu: Marching orders
        R10000 / R10010: CPU / FPU
        64: Number of CPUs
        250: Clock frequency (MHz.)

Experiment notes--
    From file L2_cache_trash.usertime.m754877:
    Caliper point 0 at target begin, PID 754877
    /hb/schang/OPTIMIZATION/dir.ssrn/L2_cache_trash
    Caliper point 1 at exit(0)

Summary of statistical callstack sampling data (usertime)--
    97: Total Samples
    0: Samples with incomplete traceback
    2.910: Accumulated Time (secs.)
    30.0: Sample interval (msecs.)

Function list, in descending order by exclusive time
-----
[index]  excl.secs  excl.%  cum.%  incl.secs  incl.%  samples  procedure  (dso: file, line)

[1]      2.550    87.6%   87.6%   2.910    100.0%   97      l2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 1)
[4]      0.300    10.3%   97.9%   0.300    10.3%   10      _RANF_4 (libfortran.so: random.c, 154)
[6]      0.030     1.0%   99.0%   0.030     1.0%    1       _xstat (libc.so.1: xstat.s, 12)
[7]      0.030     1.0%   100.0%  0.030     1.0%    1       _write (libc.so.1: write.s, 20)
[2]      0.000     0.0%   100.0%  2.910    100.0%   97      __start (L2_cache_trash: crtltxt.s, 103)
[3]      0.000     0.0%   100.0%  2.910    100.0%   97      main (libftn.so: main.c, 76)
[5]      0.000     0.0%   100.0%  0.060     2.1%    2       _FWU (libfortran.so: wu90.c, 47)
[8]      0.000     0.0%   100.0%  0.030     1.0%    1       _ll_implicit_open (libfortran.so: impopen.c, 42)

Butterfly function list, in descending order by inclusive time
-----
            attrib.%  attrib.time  self%  self-time  incl.time  caller (callsite) [index]
[index]  incl.%  incl.time  attrib.%  attrib.time  incl.time  callee (callsite) [index]

            100.0%    2.910
[1]  100.0%    2.910    87.6%    2.550    2.910  main (@0x0af99b20; libftn.so: main.c, 97) [3]
            4.1%    0.120
            3.1%    0.090
            3.1%    0.090
            2.1%    0.060
            0.000
[2]  100.0%    2.910    100.0%    2.910    2.910  __start [2]
            2.910  main (@0x10001738; L2_cache_trash: crtltxt.s, 177) [3]
            2.910
[3]  100.0%    2.910    0.0%    0.000    2.910  __start (@0x10001738; L2_cache_trash: crtltxt.s, 177) [2]
            main [3]
            2.910  l2_cache_trashing (@0x0af99b20; libftn.so: main.c, 97) [1]
            3.1%    0.090
            3.1%    0.090
            4.1%    0.120
[4]  10.3%    0.300    10.3%    0.300    2.910  l2_cache_trashing (@0x10001878; L2_cache_trash: L2_cache_trash.f, 12) [1]
            2.910  l2_cache_trashing (@0x10001810; L2_cache_trash: L2_cache_trash.f, 11) [1]
            2.910  l2_cache_trashing (@0x100017a8; L2_cache_trash: L2_cache_trash.f, 10) [1]
            _RANF_4 [4]
            2.910  l2_cache_trashing (@0x10001ac4; L2_cache_trash: L2_cache_trash.f, 20) [1]
[5]  2.1%    0.060    0.0%    0.000    2.910  _FWU [5]
            1.0%    0.030
            1.0%    0.030
            0.030  _xfer_iolist (@0x0a6e1484; libfortran.so: wu90.c, 183) [11]
            0.030  _ll_implicit_open (@0x0a6e1550; libfortran.so: wu90.c, 96) [8]
            1.0%    0.030
[6]  1.0%    0.030    1.0%    0.030    0.030  _stat (@0x0fa3d490; libc.so.1: stat.c, 32) [10]
            _xstat [6]
            1.0%    0.030
[7]  1.0%    0.030    1.0%    0.030    0.030  _write (@0x0fa4f4dc; libc.so.1: writeSCI.c, 29) [16]
            _write [7]
            1.0%    0.030
            0.060  _FWU (@0x0a6e1550; libfortran.so: wu90.c, 96) [5]

```

[8]	1.0%	0.030	0.0%	0.000	__ll_implicit_open [8]
			1.0%	0.030	_f_open (@0x0a6c6e0c; libfortran.so: impopen.c, 85) [9]

DSO List, in order of loading					

DSO name	instrs.	functions	files	lines	low - high text addr., DSO pathname
L2_cache_trash	339	6	5	0 0x100015cc - 0x10001b18	L2_cache_trash
libss.so	672	34	5	0 0x09db11f8 - 0x09db1c78	/usr/lib32/libss.so
libssrt.so	209976	2174	435	0 0x09e62fe8 - 0x09f300c8	/usr/lib32/libssrt.so [ignored]
libfortran.so	300668	1501	502	0 0x0a6b3d98 - 0x0a7d9788	/opt/MIPSPRO/MIPSPRO/usr/lib32/mips4/libfortran.so
libffio.so	117248	644	228	0 0x0a77d0d8 - 0x0a7ef8d8	/opt/MIPSPRO/MIPSPRO/usr/lib32/mips4/libffio.so
libftn.so	88249	1475	531	0 0x0af75c00 - 0x0afcbec4	/opt/MIPSPRO/MIPSPRO/usr/lib32/mips4/libftn.so
libm.so	48157	163	148	0 0x09fa4e8c - 0x09fd3f00	/opt/MIPSPRO/MIPSPRO/usr/lib32/mips4/libm.so
libc.so.1	244378	2891	962	0 0x0fa372c4 - 0x0fb25d2c	/usr/lib32/libc.so.1

□ Example : prof -l L2_cache_trash.fpcsamp.m755026

SpeedShop profile listing generated Fri Jan 26 14:58:10 2001					
prof -l L2_cache_trash.fpcsamp.m755026					
L2_cache_trash (n32): Target program					
fpcsamp: Experiment name					
pc,2,1000,0:cu: Marching orders					
R10000 / R10010: CPU / FPU					
64: Number of CPUs					
250: Clock frequency (MHz.)					
Experiment notes--					
From file L2_cache_trash.fpcsamp.m755026:					
Caliper point 0 at target begin, PID 755026					
/hb/schang/OPTIMIZATION/dir.ssrn/L2_cache_trash					
Caliper point 1 at exit(0)					

Summary of statistical PC sampling data (fpcsamp)--					
2922: Total samples					
2.922: Accumulated time (secs.)					
1.0: Time per sample (msecs.)					
2: Sample bin width (bytes)					

Function list, in descending order by time					

[index]	secs	%	cum.%	samples	function (dso: file, line)
[1]	2.451	83.9%	83.9%	2451	L2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 1)
[2]	0.370	12.7%	96.5%	370	_RANF_4 (libfortran.so: random.c, 154)
[3]	0.090	3.1%	99.6%	90	__write (libc.so.1: write.s, 20)
[4]	0.010	0.3%	100.0%	10	_ftruncate (libc.so.1: ftruncate.s, 16)
[5]	0.001	0.0%	100.0%	1	_lae_get_assign_file_name (libffio.so: asnenv.c, 606)
	2.922	100.0%	100.0%	2922	TOTAL

Line list, in descending order by function-time and then line number					

	secs	%	cum.%	samples	function (dso: file, line)
	0.051	1.7	1.7	51	L2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 10)
	0.020	0.7	2.4	20	L2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 11)
	0.041	1.4	3.8	41	L2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 12)
	2.339	80.0	83.9	2339	L2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 16)
	0.035	1.2	85.1	35	_RANF_4 (libfortran.so: random.c, 154)
	0.191	6.5	91.6	191	_RANF_4 (libfortran.so: random.c, 158)
	0.144	4.9	96.5	144	_RANF_4 (libfortran.so: random.c, 160)
	0.090	3.1	99.6	90	__write (libc.so.1: write.s, 20)
	0.010	0.3	100.0	10	_ftruncate (libc.so.1: ftruncate.s, 16)
	0.001	0.0	100.0	1	_lae_get_assign_file_name (libffio.so: asnenv.c, 672)

□ Example : prof -h L2_cache_trash.fpcsamp.m755026

```
-----
SpeedShop profile listing generated Fri Jan 26 17:53:03 2001
prof -h L2_cache_trash.fpcsamp.m755026
  L2_cache_trash (n32): Target program
    fpcsamp: Experiment name
    pc,2,1000,0:cu: Marching orders
    R10000 / R10010: CPU / FPU
    64: Number of CPUs
    250: Clock frequency (MHz.)

Experiment notes--
  From file L2_cache_trash.fpcsamp.m755026:
  Caliper point 0 at target begin, PID 755026
  /hb/schang/OPTIMIZATION/dir.ssrn/L2_cache_trash
  Caliper point 1 at exit(0)
-----
Summary of statistical PC sampling data (fpcsamp)--
  2922: Total samples
  2.922: Accumulated time (secs.)
  1.0: Time per sample (msecs.)
  2: Sample bin width (bytes)
-----
Function list, in descending order by time
-----
[index]      secs      %      cum.%      samples  function (dso: file, line)

  [1]      2.451    83.9%    83.9%      2451    l2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 1)
  [2]      0.370    12.7%    96.5%       370    _RANF_4 (libfortran.so: random.c, 154)
  [3]      0.090     3.1%    99.6%        90    __write (libc.so.1: write.s, 20)
  [4]      0.010     0.3%   100.0%        10    _ftruncate (libc.so.1: ftruncate.s, 16)
  [5]      0.001     0.0%   100.0%         1    _lae_get_assign_file_name (libffio.so: asnenv.c, 606)

      2.922  100.0%  100.0%      2922    TOTAL
-----
Line list, in descending order by time
-----
      secs      %      cum.%      samples  function (dso: file, line)

    2.339    80.0    80.0      2339    l2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 16)
    0.191     6.5    86.6       191    _RANF_4 (libfortran.so: random.c, 158)
    0.144     4.9    91.5       144    _RANF_4 (libfortran.so: random.c, 160)
    0.090     3.1    94.6        90    __write (libc.so.1: write.s, 20)
    0.051     1.7    96.3        51    l2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 10)
    0.041     1.4    97.7        41    l2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 12)
    0.035     1.2    98.9        35    _RANF_4 (libfortran.so: random.c, 154)
    0.020     0.7    99.6        20    l2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 11)
    0.010     0.3   100.0        10    _ftruncate (libc.so.1: ftruncate.s, 16)
    0.001     0.0   100.0         1    _lae_get_assign_file_name (libffio.so: asnenv.c, 672)
-----
```

□ Example : prof -b -basicblocks -q 8 L2_cache_trash.ideal.m1838173

```
-----
SpeedShop profile listing generated Tue Jan 30 13:32:42 2001
prof -b -basicblocks -q 8 L2_cache_trash.ideal.m1838173
  L2_cache_trash (n32): Target program
    ideal: Experiment name
    it:cu: Marching orders
    R10000 / R10010: CPU / FPU
    64: Number of CPUs
    250: Clock frequency (MHz.)

Experiment notes--
  From file L2_cache_trash.ideal.m1838173:
  Caliper point 0 at target begin, PID 1838173
  L2_cache_trash
  Caliper point 1 at exit(0)
-----
Summary of ideal time data (ideal)--
  162717775: Total number of instructions executed
  207314320: Total computed cycles
  0.829: Total computed execution time (secs.)
  1.274: Average cycles / instruction
-----
Function list, in descending order by exclusive ideal time
-----
[index]      excl.secs  excl.%      cum.%      cycles  instructions  incl.secs  incl.%      calls  function (dso: file, line)

  [3]      0.478     57.7%     57.7%    119572538  115374161      0.818     98.7%        1    l2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 1)
  [4]      0.340     41.0%     98.6%    84934656   44040192      0.340     41.0%    3145728    _RANF_4 (libfortran.so: random.c, 154)
  [9]      0.004      0.5%     99.1%    1019750    1009821      0.005     0.7%       3110    general_find_symbol (rld: rld.c, 2038)
 [10]      0.002      0.2%     99.4%    466070     535207      0.002     0.3%       6074    resolve_relocations (rld: rld.c, 2636)
 [12]      0.001      0.1%     99.5%    266235     335241      0.001     0.1%       3124    elfhash (rld: obj.c, 1184)
 [13]      0.001      0.1%     99.6%    141808     203673      0.001     0.1%       8863    obj_dynsym_got (rld: objfcn.c, 46)
  [8]      0.001      0.1%     99.6%    139493     176748      0.006     0.7%       3065    resolve_symbol (rld: rld.c, 1828)
  [7]      0.001      0.1%     99.7%    136156     176862      0.008     0.9%       3110    resolving (rld: rld.c, 1499)
-----
Butterfly function list, in descending order by inclusive ideal time
-----
      attrib.%  attrib.time(#calls)      incl.time  caller (callsite) [index]
[index]  incl.%  incl.time      self%  self-time  procedure [index]
      attrib.%  attrib.time(#calls)      incl.time  callee (callsite) [index]

  [1]    98.7%    0.818      0.0%    0.000    __start [1]
      98.7%    0.818(0000001)      0.818    main [2]
      0.0%    0.000(0000001)      0.000    __readenv_sigfpe [316]
      0.0%    0.000(0000001)      0.000    __istart [320]
-----
  [2]    98.7%    0.818(0000001)      0.818    __start [1]
      98.7%    0.818      0.0%    0.000    main [2]
      98.7%    0.818(0000001)      0.818    l2_cache_trashing [3]
      0.0%    0.000(0000005)      0.000    signal [146]
-----
  [3]    98.7%    0.818(0000001)      0.818    main [2]
      98.7%    0.818      57.7%    0.478    l2_cache_trashing [3]
      41.0%    0.340(3145728)      0.340    _RANF_4 [4]
      0.0%    0.000(0000001)      0.000    _FWU [42]
      0.0%    0.000(0000001)      0.000    _F90_STOP [113]
-----
  [4]    41.0%    0.340(3145728)      0.340    l2_cache_trashing [3]
      41.0%    0.340      41.0%    0.340    _RANF_4 [4]
-----
  [5]    1.2%     0.010      0.0%    0.000    handle_undefineds [5]
      1.0%    0.008(0000008)      0.008    search_for_externals [6]
      0.2%    0.001(0000003)      0.001    fix_all_defineds [11]
      0.0%    0.000(0000001)      0.000    search_for_undefineds [93]
-----
  [6]    1.0%    0.008(0000008)      0.010    handle_undefineds [5]
      1.0%    0.008      0.0%    0.000    search_for_externals [6]
      0.9%    0.008(0003109)      0.008    resolving [7]
      0.0%    0.000(0000452)      0.000    do_relocations_and_check_reloc_on_stub [17]
      0.0%    0.000(0000467)      0.001    obj_dynsym_got [13]
-----
http://www.nas.nasa.gov/~schang/origin_opt.html
```

	0.0%	0.000(0000001)		0.000	search_for_undefineds [93]
	0.9%	0.008(0003109)		0.008	search_for_externals [6]
[7]	0.9%	0.008	0.1%	0.001	resolving [7]
			0.7%	0.006(0003026)	0.006 resolve_symbol [8]
			0.1%	0.001(0003110)	0.002 resolve_relocations [10]
			0.0%	0.000(0003110)	0.000 obj_set_dynsym_got [18]
			0.0%	0.000(0000096)	0.000 common_handling [51]
			0.0%	0.000(0000024)	0.000 get_symtab_table_entry [178]
			0.0%	0.000(0000024)	0.000 is_optional_symbol [187]

	0.0%	0.000(0000010)		0.000	_rld_name_to_address [68]
	0.0%	0.000(0000029)		0.000	lazy_text_resolve [38]
	0.7%	0.006(0003026)		0.008	resolving [7]
[8]	0.7%	0.006	0.1%	0.001	resolve_symbol [8]
			0.6%	0.005(0003062)	0.005 general_find_symbol [9]
			0.0%	0.000(0000008)	0.000 common_handling [51]
			0.0%	0.000(0000003)	0.000 strcmp [14]

Basic block list, in descending order by total cycles					

index	cyc./cnt.	counts	total cycles	%	function (address; dso: file, line)
0	27	3145728	84934656	41.0%	_RANF_4 (@0x0a7132d0; libfortran.so: random.c, 154)
1	57	1048576	59768832	28.8%	L2_cache_trashing (@0x100018e4; L2_cache_trash: L2_cache_trash.f, 16)
2	14	1048576	14680064	7.1%	L2_cache_trashing (@0x100017b0; L2_cache_trash: L2_cache_trash.f, 10)
3	14	1048576	14680064	7.1%	L2_cache_trashing (@0x10001818; L2_cache_trash: L2_cache_trash.f, 11)
4	14	1048576	14680064	7.1%	L2_cache_trashing (@0x10001880; L2_cache_trash: L2_cache_trash.f, 12)
5	5	1048576	5242880	2.5%	L2_cache_trashing (@0x10001874; L2_cache_trash: L2_cache_trash.f, 12)
6	5	1048576	5242880	2.5%	L2_cache_trashing (@0x1000180c; L2_cache_trash: L2_cache_trash.f, 11)
7	5	1048576	5242880	2.5%	L2_cache_trashing (@0x100017a4; L2_cache_trash: L2_cache_trash.f, 10)

□ Example : prof -l L2_cache_trash.dsc_hwc.m1225967

```

SpeedShop profile listing generated Fri Jan 26 14:46:26 2001
prof -l lomax.L2_cache_trash.dsc_hwc.m1225967
  L2_cache_trash (n32): Target program
    dsc_hwc: Experiment name
      hwc,26,131:cu: Marching orders
    R8000 / Unknown chip: CPU / FPU
      0: Number of CPUs
      250: Clock frequency (MHz.)

Experiment notes--
  From file lomax.L2_cache_trash.dsc_hwc.m1225967:
  Caliper point 0 at target begin, PID 1225967
    L2_cache_trash
  Caliper point 1 at exit(0)

Summary of perf. counter overflow PC sampling data (dsc_hwc)--
  31759: Total samples
  Secondary cache D misses (26): Counter name (number)
    131: Counter overflow value
  4160429: Total counts
-----
Function list, in descending order by counts
[index]      counts      %    cum.%    samples  function (dso: file, line)

[1]          4159250 100.0% 100.0%    31750    L2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 1)
[2]          131      0.0% 100.0%         1    _ns_lookup (libc.so.1: ns_lookup.c, 30)
[3]          131      0.0% 100.0%         1    memset (libc.so.1: bzero.s, 98)
[4]          131      0.0% 100.0%         1    __filbuf (libc.so.1: __filbuf.c, 27)
[5]          131      0.0% 100.0%         1    _unit_close (libfortran.so: unitclose.c, 53)
           655      0.0% 100.0%         5    **OTHER** (includes excluded DSOs, rld, etc.)

          4160429 100.0% 100.0%    31759    TOTAL
-----
Line list, in descending order by function-time and then line number
counts      %    cum.%    samples  function (dso: file, line)

    131      0.0    0.0         1    L2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 11)
    131      0.0    0.0         1    L2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 12)
  4158988  100.0  100.0    31748    L2_cache_trashing (L2_cache_trash: L2_cache_trash.f, 16)

    131      0.0  100.0         1    __filbuf (libc.so.1: __filbuf.c, 39)

    131      0.0  100.0         1    _ns_lookup (libc.so.1: ns_lookup.c, 45)

    131      0.0  100.0         1    memset (libc.so.1: bzero.s, 133)

    131      0.0  100.0         1    _unit_close (libfortran.so: unitclose.c, 138)

```

□ Example : Traces floating point exceptions

When the following program is compiled with default,i.e., f90 -o fpe.fpe, array a is of type real*4. Thus the allowed range of a(i) is approximately -10**(-38).LE.10**(38) on IRIX systems. In the following example, a(37)-a(100) are reported to be infinity in the output and thus the operation in this program caused floating point exception 64 times. ssrun -fpe reports 65 (?) counts of FPEs.

Sample program : fpe.f

```

program fpe
dimension a(100)
b=0.1
do i=1,100
b=b*0.1
a(i)=1.0/b
end do

write (6,*) a

stop
end

```

prof floating_point_exe.fpe.m2688278

```
-----
SpeedShop profile listing generated Tue Feb  6 10:52:23 2001
prof floating_point_exe.fpe.m2688278
floating_point_exe (n32): Target program
                        fpe: Experiment name
                        fpe:cu: Marching orders
                        R10000 / R10010: CPU / FPU
                        64: Number of CPUs
                        250: Clock frequency (MHz.)
Experiment notes--
  From file floating_point_exe.fpe.m2688278:
  Caliper point 0 at target begin, PID 2688278
    /scratch1/schang/OPTIMIZATION/dir.ssrn/floating_point_exe
  Caliper point 1 at exit(0)
-----
Summary of FPE callstack tracing data (fpe)--
                        65: Total FPEs
                        0: Samples with incomplete traceback
-----
Function list, in descending order by exclusive FPEs
-----
[index]  excl.FPEs excl.%  cum.%  incl.FPEs incl.%  function (dso: file, line)
-----
[1]      65 100.0%  100.0%      65 100.0%  fpe (floating_point_exe: fpe.f, 1)
[2]      0   0.0%  100.0%      65 100.0%  __start (floating_point_exe: crt1text.s, 103)
[3]      0   0.0%  100.0%      65 100.0%  main (libftn.so: main.c, 76)
-----
```

o pixie

- Pixie is a tool used to measure execution frequency of each basic block in a program
- It is normally invoked from **ssrun** with the -ideal option. See [PBS script for ssrun](#) and [ssrun -ideal](#) examples to learn how to do this.
- It can be directly invoked by a user without ssrun.

□ Command

pixie executable_name [pixie-options]

Only two of these options are described here. Please refer to pixie man page for a complete listing.

```
-pixie_file
Specify the name of the pixiefied executable.
Default: append ".pixie" (or an explicit suffix, as set by -suffix)
to the original name.

-counts_file
Specify the name to be used for the output .Counts file.
Default: append ".Counts" to the original program name.
```

Example : Use pixie directly to count the execution frequency of each basic block

```
module load SpeedShop.1.4 <--- This is currently required on Hopper
/bin/time pixie L2_cache_trash -pixie_file L2_cache_trash.pixie -counts_file L2_cache_trash.Counts
/bin/time L2_cache_trash.pixie <--- This step generates the counts_file
```

In this example, L2_cache_trash is the original executable, and L2_cache_trash.pixie is the pixiefied executable to be generated by pixie. L2_cache_trash.Counts is the chosen file name for the counts_file. This file won't be generated until the pixiefied executable is invoked.

□ Use prof to analyze pixie counts_file and generate a report

Example : prof -b -basicblocks -q 8 L2_cache_trash.Counts > pixie.out

```

-----
SpeedShop profile listing generated Tue Feb  6 12:53:32 2001
prof -b -basicblocks -q 8 L2_cache_trash.Counts
    L2_cache_trash (n32): Target program
        pixie-counts: Experiment name
        pixie-counts: Marching orders
        R10000 / R10010: CPU / FPU
        64: Number of CPUs
        250: Clock frequency (MHz.)
-----
Summary of ideal time data (pixie-counts)--
    162404649: Total number of instructions executed
    207035122: Total computed cycles
    0.828: Total computed execution time (secs.)
    1.275: Average cycles / instruction
-----
Function list, in descending order by exclusive ideal time
-----
[index]  excl.secs  excl.%  cum.%  cycles  instructions  incl.secs  incl.%  calls  function (dso: file, line)
-----
_RANF_4 (libfortran.so: random.c, 154)
  [9]    0.003    0.4%    99.2%    840687    830150    0.005    0.6%    3084  general_find_symbol (rld: rld.c, 2038)
 [10]    0.002    0.2%    99.4%    464987    533835    0.002    0.3%    6039  resolve_relocations (rld: rld.c, 2636)
 [12]    0.001    0.1%    99.5%    262250    330258    0.001    0.1%    3089  elfhash (rld: obj.c, 1184)
 [13]    0.001    0.1%    99.6%    140800    202224    0.001    0.1%    8800  obj_dynsym_got (rld: objfcn.c, 46)
  [8]    0.001    0.1%    99.7%    138415    175368    0.005    0.6%    3038  resolve_symbol (rld: rld.c, 1828)
  [7]    0.001    0.1%    99.7%    135452    175950    0.007    0.9%    3094  resolving (rld: rld.c, 1499)
-----
Butterfly function list, in descending order by inclusive ideal time
-----
      attrib.%  attrib.time(#calls)  self-time  procedure [index]  incl.time  caller (callsite) [index]
[index]  incl.%  incl.time  attrib.%  attrib.time(#calls)  incl.time  callee (callsite) [index]
-----
  [1]  98.8%    0.818    0.0%    0.000    __start [1]    0.818  main [2]
      98.8%    0.818(00000001)    0.0%    0.000(00000001)    0.000  __readenv_sigfpe [286]
      0.0%    0.000(00000001)    0.000  __istart [289]
-----
  [2]  98.8%    0.818(00000001)    0.0%    0.000  main [2]    0.818  __start [1]
      98.8%    0.818    0.0%    0.818(00000001)    0.818  l2_cache_trashing [3]
      0.0%    0.000(00000005)    0.000  signal [118]
-----
  [3]  98.8%    0.818(00000001)    57.8%    0.478  l2_cache_trashing [3]    0.818  main [2]
      98.8%    0.818    41.0%    0.340(3145728)    0.340  _RANF_4 [4]
      0.0%    0.000(00000001)    0.000  _FWU [32]
      0.0%    0.000(00000001)    0.000  _F90_STOP [92]
-----
  [4]  41.0%    0.340(3145728)    41.0%    0.340  _RANF_4 [4]    0.818  l2_cache_trashing [3]
      41.0%    0.340    0.0%    0.000
-----
  [5]  1.1%    0.009    0.0%    0.000  handle_undefineds [5]    0.008  search_for_externals [6]
      0.9%    0.008(00000006)    0.2%    0.001(00000003)    0.001  fix_all_defineds [11]
      0.0%    0.000(00000001)    0.000  search_for_undefineds [77]
-----
  [6]  0.9%    0.008(00000006)    0.0%    0.000  search_for_externals [6]    0.009  handle_undefineds [5]
      0.9%    0.008    0.9%    0.007(0003093)    0.007  resolving [7]
      0.0%    0.000(0000433)    0.000  do_relocations_and_check_reloc_on_stub [16]
      0.0%    0.000(0000438)    0.001  obj_dynsym_got [13]
-----
  [7]  0.0%    0.000(00000001)    0.0%    0.000  search_for_undefineds [77]
      0.9%    0.007(0003093)    0.0%    0.000  search_for_externals [6]
      0.0%    0.000    0.1%    0.001  resolving [7]
      0.6%    0.005(0003010)    0.005  resolve_symbol [8]
      0.1%    0.001(0003094)    0.002  resolve_relocations [10]
      0.0%    0.000(0003094)    0.000  obj_set_dynsym_got [15]
      0.0%    0.000(0000096)    0.000  common_handling [40]
      0.0%    0.000(0000024)    0.000  get_symlib_table_entry [154]
      0.0%    0.000(0000024)    0.000  is_optional_symbol [164]
-----
  [8]  0.0%    0.000(00000001)    0.0%    0.000  _rld_name_to_address [123]
      0.0%    0.000(0000027)    0.000  lazy_text_resolve [30]
      0.6%    0.005(0003010)    0.007  resolving [7]
      0.0%    0.000    0.1%    0.001  resolve_symbol [8]
      0.6%    0.005(0003036)    0.005  general_find_symbol [9]
      0.0%    0.000(0000005)    0.000  common_handling [40]
      0.0%    0.000(0000002)    0.000  strcmp [14]
-----
Basic block list, in descending order by total cycles
-----
      index  cyc./cnt.  counts  total cycles  %  function (address; dso: file, line)
-----
      0      27      3145728    84934656    41.0%  _RANF_4 (@0x0a7132d0; libfortran.so: random.c, 154)
      1      57      1048576    59768832    28.9%  l2_cache_trashing (@0x100018e4; L2_cache_trash: L2_cache_trash.f, 16)
      2      14      1048576    14680064    7.1%  l2_cache_trashing (@0x10001818; L2_cache_trash: L2_cache_trash.f, 11)
      3      14      1048576    14680064    7.1%  l2_cache_trashing (@0x10001880; L2_cache_trash: L2_cache_trash.f, 12)
      4      14      1048576    14680064    7.1%  l2_cache_trashing (@0x100017b0; L2_cache_trash: L2_cache_trash.f, 10)
      5       5      1048576    5242880    2.5%  l2_cache_trashing (@0x100017a4; L2_cache_trash: L2_cache_trash.f, 10)
      6       5      1048576    5242880    2.5%  l2_cache_trashing (@0x1000180c; L2_cache_trash: L2_cache_trash.f, 11)
      7       5      1048576    5242880    2.5%  l2_cache_trashing (@0x10001874; L2_cache_trash: L2_cache_trash.f, 12)

```

Note: This output is almost identical to that one produced by the `ssrun -ideal`. The main differences occur in the header and summary sections as highlighted with blue color.

How to Proceed

● Measure the Time Performance of the Entire Code

- `/bin/time`
- `/bin/timex`
- `/bin/ssusage`

● Find Out the Sections/Routines that Use the Most Time

- `ssrun -pcsamp a.out` : sampling the address of instruction 10 ms timer
- `ssrun -fpcsamp a.out` : 1 ms timer
- `ssrun -usertime a.out` : sampling the call stack 30 ms timer
- insert `dtime`, `etime`, second or `timef` functions in source code

The output generated by `ssrun` can be analyzed with `prof`. Use the `-l` or `-h` to find the most heavily used line and use `-b` if you want to know the caller and callee.

● Find Out How Many Times Each Routine is Called and If Its Performance is Optimal

http://www.nasa.gov/~schang/origin_opt.html

- o ssrun -ideal a.out
- o pixie a.out

Both are based on 'basic block' profiling. They give exact count of the number of times each basic block in the program is entered during a run.

The output generated by ssrun or pixie can be analyzed with prof. Use the -l or -h to find the most heavily used line and use -b if you want to know the caller and callee.

If the ideal time reported by prof for a function is much smaller than the those from -usertime or -[f]pcsamp, then the performance is not optimal for this function. This function thus is a potential target to be optimized.

● Find Out What Causes the Entire Code not to Perform Well

- o perfex -a -x -y a.out

It provides information regarding which events take more clock-cycles than others. However, it does not provide information regarding which sections/routines of the code are causing the large clock-cycles for those events.

● For a Specific Hardware Performance Counter, Find Out Which Routines Contribute the Most

- o ssrun *_hwc a.out

Experiment_Type	Description
-gi_hwc and -fgi_hwc	graduated instruction; functions that burn a lot of instructions
-cy_hwc and -fcy_hwc	elapsed cycles; functions with cache misses and mispredicted branches
-ic_hwc and -fic_hwc	code (instruction) that does not fit in L1 cache
-isc_hwc and -fisc_hwc	code (instruction) that does not fit in L2 cache
-dc_hwc and -fdc_hwc	code that causes L1 data cache misses
-dsc_hwc and -fdsc_hwc	code that causes L2 data cache misses
-tlb_hwc and -ftlb_hwc	code that cuase TLB misses
-gfp_hwc and -fgfp_hwc	code that performs heavy FP calculation

- o insert libperfex calls in selected routines of source code

SINGLE-CPU OPTIMIZATION TECHNIQUES

Sources of Performance Problems

- CPU-bound processes
 - o Performing many slow operations such as sqrt, fp divides
 - o Non-pipelined operations: switching between adds and mults
- Memory-bound processes
 - o Poor memory strides
 - o Page thrashing
 - o Cache misses
- I/O bound processes
 - o Performing synchronous I/O
 - o Performing formatted I/O
 - o Library and system level buffering

Useful Techniques

● Replace Division by Multiplication

Division takes more clockcycles than multiplication. In this example, the performances of division versus multiplication are compared using dtime. The program is compiled with default, i.e., f90 -o d_to_m d_to_m.f

Sample program : d_to_m.f

```
program d_to_m

! This program demonstrates the better performance by
! using multiplication than division

      dimension a(2000,2000),b(2000,2000)
      real*4 dtime,tarray(2)

      pi=3.14159
      pinv=1./3.14159

! Division

      t1=dtime(tarray)

      do iteration=1,10
      do j=1,2000
      do i=1,2000
      a(i,j)=float(i+j)/pi
      end do
      end do
      end do

      t2=dtime(tarray)

      write (6,*) 'time spent using division = ',t2

! Multiplication

      t1=dtime(tarray)

      do iteration=1,10
      do j=1,2000
      do i=1,2000
      b(i,j)=float(i+j)*pinv
      end do
      end do
      end do

      t2=dtime(tarray)

      write (6,*) 'time spent using multiplication = ', t2

      write (12) a
      write (12) b

      stop
      end
```

Output (hopper, R10000, 250MHz)

```
time spent using division      =  5.64292097
time spent using multiplication =  4.30546093
```

● Increase Page Size to Reduce TLB Miss and Page Fault

Default and allowed page sizes on NAS machines:

Page Size	Hopper	Steger	Lomax
16KB	default	default	default
64KB	no	no	yes
256KB	no	no	yes
1MB	yes	yes	yes
4MB	no	no	yes
16MB	no	no	yes

Increase page size with:

% dplace -data_pagesize 1M -stack_pagesize 1M a.out

Example : Increasing Performance with Larger Page Size

Sample program : tlb.f

```
program tlb

! a(i,j,k) and a(i,j,k+1) are (1024*100*4B) 400KB apart.
! a(i,j,k) and b(i,j,k) are ~40MB apart.
! If the page size is 16KB,
! a(i,j,k) and a(i,j,k+1) are 25 (virtual) pages away from each other.
! a(i,j,k) and b(i,j,k) are ~ 2500 (virtual) pages away from each other.
! Thus, a(i,j,k), b(i,j,k), c(i,j,k) .. are all on different pages.
! Accessing these arrays along the outmost direction k causes
! TLB having to update pages continuously.

      dimension a(1024,100,100),b(1024,100,100),c(1024,100,100)
      @      ,d(1024,100,100),e(1024,100,100),f(1024,100,100)
      @      ,g(1024,100,100),h(1024,100,100),p(1024,100,100)
      @      ,q(1024,100,100),r(1024,100,100),s(1024,100,100)

      do i=1,1024
      do j=1,100
      do k=1,100
      a(i,j,k)=float(i+j+k)
      b(i,j,k)=2.0*float(i-j+k)
      c(i,j,k)=a(i,j,k)*b(i,j,k)
      d(i,j,k)=c(i,j,k)*a(i,j,k)
      e(i,j,k)=amax0(i,j,k)
      f(i,j,k)=e(i,j,k)*d(i,j,k)
      g(i,j,k)=log(float(i+j+k))
      h(i,j,k)=g(i,j,k)+f(i,j,k)
      p(i,j,k)=g(i,j,k)-f(i,j,k)
      q(i,j,k)=a(i,j,k)+b(i,j,k)+h(i,j,k)
      r(i,j,k)=c(i,j,k)+d(i,j,k)-e(i,j,k)
      s(i,j,k)=h(i,j,k)-a(i,j,k)
      end do
      end do
      end do

      stop
      end
```

Compile and Execute on R12000 Lomax

f90 -o tlb tlb.f

Perfex Output Using Default Page Size 16K

```
/bin/time perfex -a -x -y tlb
WARNING: Multiplexing events to project totals--inaccuracy possible
Summary for execution of tlb

Based on 400 MHz IP27
MIPS R12000 CPU

Event Counter Name                      Counter Value  Typical Time (sec)  Minimum Time (sec)  Maximum Time (sec)
=====
0 Cycles..... 35163735984  87.909340  87.909340  87.909340
16 Executed prefetch instructions..... 0  0.000000  0.000000  0.000000
4 Miss handling table occupancy..... 26559242784  66.398107  66.398107  66.398107
23 TLB misses..... 174946720  34.018390  34.018390  34.018390
26 Secondary data cache misses..... 123832880  30.924166  19.512966  30.924166
7 Quadwords written back from scache..... 984426528  20.894453  14.520291  21.583552
2 Decoded loads..... 2730838208  6.827096  6.827096  6.827096
25 Primary data cache misses..... 272815360  5.797326  1.480023  5.797326
18 Graduated loads..... 1956728688  4.891822  4.891822  4.891822
22 Quadwords written back from primary data cache..... 247412768  2.461757  1.942190  2.461757
3 Decoded stores..... 319666496  0.799166  0.799166  0.799166
6 Resolved conditional branches..... 318546432  0.796366  0.796366  0.796366
21 Graduated floating point instructions..... 297206064  0.743015  0.371508  38.636788
24 Mispredicted branches..... 40145504  0.730648  0.602183  0.884205
19 Graduated stores..... 152505072  0.381263  0.381263  0.381263
9 Primary instruction cache misses..... 714320  0.030376  0.007750  0.030376
10 Secondary instruction cache misses..... 75248  0.018791  0.011857  0.018791
30 Store/prefetch exclusive to clean block in scache..... 3136  0.000008  0.000008  0.000008
31 Store/prefetch exclusive to shared block in scache..... 528  0.000001  0.000001  0.000001
1 Decoded instructions..... 14183703344  0.000000  0.000000  35.459258
5 Failed store conditionals..... 0  0.000000  0.000000  0.000000
8 Correctable scache data array ECC errors..... 0  0.000000  0.000000  0.000000
11 Instruction misprediction from scache way prediction table.. 23760  0.000000  0.000000  0.000059
12 External interventions..... 22240  0.000000  0.000000  0.000000
13 External invalidations..... 631776  0.000000  0.000000  0.000000
14 ALU/FPU progress cycles..... 0  0.000000  0.000000  0.000000
15 Graduated instructions..... 10792080496  0.000000  0.000000  26.980201
17 Prefetch primary data cache misses..... 0  0.000000  0.000000  0.000000
20 Graduated store conditionals..... 0  0.000000  0.000000  0.000000
27 Data misprediction from scache way prediction table..... 595520  0.000000  0.000000  0.001489
28 State of intervention hits in scache..... 22128  0.000000  0.000000  0.000000
29 State of invalidation hits in scache..... 36848  0.000000  0.000000  0.000000

Statistics
=====
Graduated instructions/cycle..... 0.306909
Graduated floating point instructions/cycle..... 0.008452
Graduated loads & stores/cycle..... 0.059983
Graduated loads & stores/floating point instruction..... 7.096873
Mispredicted branches/Resolved conditional branches..... 0.126027
Graduated loads /Decoded loads ( and prefetches )..... 0.716530
Graduated stores/Decoded stores..... 0.477076
Data mispredict/Data scache hits..... 0.003997
Instruction mispredict/Instruction scache hits..... 0.037179
L1 Cache Line Reuse..... 6.731360
L2 Cache Line Reuse..... 1.203093
L1 Data Cache Hit Rate..... 0.870657
L2 Data Cache Hit Rate..... 0.546093
Time accessing memory/Total time..... 0.864675
Time not making progress (probably waiting on memory) / Total time..... 1.000000
L1-L2 bandwidth used (MB/s, average per process)..... 144.338427
Memory bandwidth used (MB/s, average per process)..... 359.477538
MFLOPS (average per process)..... 3.380825
Cache misses in flight per cycle (average)..... 0.755302
Prefetch cache miss rate..... nan0x7fffffff

real 91.517
user 87.940
sys 3.181
```

Perfex Output Using Page Size of 1MB

```

/bin/time perfex -a -x -y dplace -data_pagesize 1M -stack_pagesize 1M tlb
WARNING: Multiplexing events to project totals--inaccuracy possible
Summary for execution of dplace -data_pagesize 1M -stack_pagesize 1M tlb

Based on 400 MHz IP27
MIPS R12000 CPU
Event Counter Name          Counter Value    Typical
                               Time (sec)      Time (sec)      Minimum
                               Time (sec)      Time (sec)      Maximum
=====
0 Cycles.....                28095818368      70.239546      70.239546      70.239546
16 Executed prefetch instructions.....                0      0.000000      0.000000      0.000000
4 Miss handling table occupancy.....            32025881120      80.064703      80.064703      80.064703
26 Secondary data cache misses.....            123657536      30.880378      19.485336      30.880378
7 Quadwords written back from scache.....            983304912      20.870647      14.503747      21.558960
23 TLB misses.....            41087280      7.989422      7.989422      7.989422
25 Primary data cache misses.....            287254448      6.104157      1.558355      6.104157
2 Decoded loads.....            1771502816      4.428757      4.428757      4.428757
18 Graduated loads.....            1549964112      3.874910      3.874910      3.874910
22 Quadwords written back from primary data cache.....            246972096      2.457372      1.938731      2.457372
21 Graduated floating point instructions.....            296999248      0.742498      0.371249      38.609902
3 Decoded stores.....            219879392      0.549698      0.549698      0.549698
19 Graduated stores.....            169628160      0.424070      0.424070      0.424070
6 Resolved conditional branches.....            150019136      0.375048      0.375048      0.375048
24 Mispredicted branches.....            14791824      0.269211      0.221877      0.325790
10 Secondary instruction cache misses.....            73840      0.018440      0.011635      0.018440
9 Primary instruction cache misses.....            368592      0.015674      0.003999      0.015674
30 Store/prefetch exclusive to clean block in scache.....            1792      0.000004      0.000004      0.000004
31 Store/prefetch exclusive to shared block in scache.....            304      0.000001      0.000001      0.000001
1 Decoded instructions.....            9803036800      0.000000      0.000000      24.507592
5 Failed store conditionals.....                0      0.000000      0.000000      0.000000
8 Correctable scache data array ECC errors.....                0      0.000000      0.000000      0.000000
11 Instruction misprediction from scache way prediction table..            29632      0.000000      0.000000      0.000000
12 External interventions.....            16096      0.000000      0.000000      0.000000
13 External invalidations.....            364560      0.000000      0.000000      0.000000
14 ALU/FPU progress cycles.....                0      0.000000      0.000000      0.000000
15 Graduated instructions.....            8817825584      0.000000      0.000000      22.044564
17 Prefetch primary data cache misses.....                0      0.000000      0.000000      0.000000
20 Graduated store conditionals.....                0      0.000000      0.000000      0.000000
27 Data misprediction from scache way prediction table.....            571008      0.000000      0.000000      0.001428
28 State of intervention hits in scache.....            16096      0.000000      0.000000      0.000000
29 State of invalidation hits in scache.....            29280      0.000000      0.000000      0.000000

Statistics
=====
Graduated instructions/cycle.....            0.313848
Graduated floating point instructions/cycle.....            0.010571
Graduated loads & stores/cycle.....            0.061205
Graduated loads & stores/floating point instruction.....            5.789888
Mispredicted branches/Resolved conditional branches.....            0.098600
Graduated loads /Decoded loads ( and prefetches ).....            0.874943
Graduated stores/Decoded stores.....            0.771460
Data mispredict/Data scache hits.....            0.003490
Instruction mispredict/Instruction scache hits.....            0.100532
L1 Cache Line Reuse.....            4.986303
L2 Cache Line Reuse.....            1.322984
L1 Data Cache Hit Rate.....            0.832952
L2 Data Cache Hit Rate.....            0.569519
Time accessing memory/Total time.....            0.701499
Time not making progress (probably waiting on memory) / Total time.....            1.000000
L1-L2 bandwidth used (MB/s, average per process).....            187.126720
Memory bandwidth used (MB/s, average per process).....            449.334385
MFLOPS (average per process).....            4.228377
Cache misses in flight per cycle (average).....            1.139881
Prefetch cache miss rate.....            nan0x7fffffff

real 73.264
user 70.302 <-- improved from ~88 to 70 seconds
sys 2.677

```

• Interchange Loops to Improve Cache Utilization

tlb.f shown above suffers tlb misses and secondary cache misses. By interchanging the loop order for operation such that the inner loop walks through the first index (i) of arrays instead of the last index (k) of the arrays reduces tlb misses and secondary cache misses tremendously as demonstrated in the perfex output.

Sample program : loop_interchange.f

```

program loop_interchange
dimension a(1024,100,100),b(1024,100,100),c(1024,100,100)
@ ,d(1024,100,100),e(1024,100,100),f(1024,100,100)
@ ,g(1024,100,100),h(1024,100,100),p(1024,100,100)
@ ,q(1024,100,100),r(1024,100,100),s(1024,100,100)

! this program is modified from tlb.f
! changing loop order to access array elements along the inner most
! direction improves performance since
! a(i,j,k) and a(i+1,j,k) are only 4B apart, i.e., a stride-one access
! Accessing a(i,j,k), b(i,j,k), c(i,j,k) will not cause bad performance
! as long as the pages containing them are already available in TLB

! do i=1,1024      !used in tlb.f
! do j=1,100       !used in tlb.f
do k=1,100
do j=1,100      !used in this program
do i=1,1024    !used in this program
a(i,j,k)=float(i+j+k)
b(i,j,k)=2.0*float(i-j+k)
c(i,j,k)=a(i,j,k)*b(i,j,k)
d(i,j,k)=c(i,j,k)*a(i,j,k)
e(i,j,k)=amax0(i,j,k)
f(i,j,k)=e(i,j,k)*d(i,j,k)
g(i,j,k)=log(float(i+j+k))
h(i,j,k)=g(i,j,k)+f(i,j,k)
p(i,j,k)=g(i,j,k)-f(i,j,k)
q(i,j,k)=a(i,j,k)+b(i,j,k)+h(i,j,k)
r(i,j,k)=c(i,j,k)+d(i,j,k)-e(i,j,k)
s(i,j,k)=h(i,j,k)-a(i,j,k)
end do
end do
end do

stop
end

```

perfex -a -x -y output (lomapx)

```

/bin/time perfix -a -x -y loop_interchange
WARNING: Multiplexing events to project totals--inaccuracy possible
Summary for execution of loop_interchange

Based on 400 MHz IP27
MIPS R12000 CPU
Typical
Event Counter Name      Counter Value  Time (sec)  Minimum  Maximum
                        Time (sec)  Time (sec)  Time (sec)
=====
0 Cycles.....          5102914448  12.757286  12.757286  12.757286
16 Executed prefetch instructions.....          0  0.000000  0.000000  0.000000
4 Miss handling table occupancy.....      2981863952  7.454660  7.454660  7.454660
25 Primary data cache misses.....      221214752  4.700813  1.200090  4.700813
2 Decoded loads.....      1428923792  3.572309  3.572309  3.572309
18 Graduated loads.....      1356947792  3.392369  3.392369  3.392369
22 Quadwords written back from primary data cache.....      253272672  2.520063  1.988190  2.520063
21 Graduated floating point instructions.....      293626704  0.734067  0.367033  38.171472
3 Decoded stores.....      138934400  0.347336  0.347336  0.347336
19 Graduated stores.....      133849120  0.334623  0.334623  0.334623
6 Resolved conditional branches.....      49797040  0.124493  0.124493  0.124493
9 Primary instruction cache misses.....      346272  0.014725  0.003757  0.014725
26 Secondary data cache misses.....      33328  0.008323  0.005252  0.008323
23 TLB misses.....      27376  0.005323  0.005323  0.005323
7 Quadwords written back from scache.....      87392  0.001855  0.001289  0.001916
24 Mispredicted branches.....      39216  0.000714  0.000588  0.000864
10 Secondary instruction cache misses.....      1168  0.000292  0.000184  0.000292
31 Store/prefetch exclusive to shared block in scache.....      128  0.000000  0.000000  0.000000
30 Store/prefetch exclusive to clean block in scache.....      32  0.000000  0.000000  0.000000
1 Decoded instructions.....      7419573696  0.000000  0.000000  18.548934
5 Failed store conditionals.....          0  0.000000  0.000000  0.000000
8 Correctable scache data array ECC errors.....          0  0.000000  0.000000  0.000000
11 Instruction misprediction from scache way prediction table..      17392  0.000000  0.000000  0.000000
12 External interventions.....      5760  0.000000  0.000000  0.000000
13 External invalidations.....      1160384  0.000000  0.000000  0.000000
14 ALU/FPU progress cycles.....          0  0.000000  0.000000  0.000000
15 Graduated instructions.....      7128009536  0.000000  0.000000  17.820024
17 Prefetch primary data cache misses.....          0  0.000000  0.000000  0.000000
20 Graduated store conditionals.....          0  0.000000  0.000000  0.000000
27 Data misprediction from scache way prediction table.....      826560  0.000000  0.000000  0.002066
28 State of intervention hits in scache.....      5760  0.000000  0.000000  0.000000
29 State of invalidation hits in scache.....      10832  0.000000  0.000000  0.000000

Statistics
=====
Graduated instructions/cycle.....          1.396851
Graduated floating point instructions/cycle.....          0.057541
Graduated loads & stores/cycle.....          0.292146
Graduated loads & stores/floating point instruction.....          5.077184
Mispredicted branches/Resolved conditional branches.....          0.000788
Graduated loads /Decoded loads ( and prefetches ).....          0.949629
Graduated stores/Decoded stores.....          0.963398
Data mispredict/Data scache hits.....          0.003737
Instruction mispredict/Instruction scache hits.....          0.050396
L1 Cache Line Reuse.....          5.739139
L2 Cache Line Reuse.....          6636.504561
L1 Data Cache Hit Rate.....          0.851613
L2 Data Cache Hit Rate.....          0.999849
Time accessing memory/Total time.....          0.661697
Time not making progress (probably waiting on memory) / Total time.....          1.000000
L1-L2 bandwidth used (MB/s, average per process).....          872.539403
Memory bandwidth used (MB/s, average per process).....          0.444002
MFLOPS (average per process).....          23.016392
Cache misses in flight per cycle (average).....          0.584345
Prefetch cache miss rate.....          nan0x7fffffff

real 16.476
user 12.724 <--- big improvement from ~87 seconds
sys 3.577

```

● Group Data Used at the Same Time to Reduce Traffic to Cache

If the access of array elements is not sequential, but indirect through an index array, index(i) as shown in the following example, it is not likely that the accesses are stride-one from iteration to iteration. Thus each iteration of the loop may cause new cache lines (3 lines in the example) to be loaded. If for each value of $j = \text{index}(i)$, $x(j)$, $y(j)$, and $z(j)$ are always processed together, then by grouping $x(j)$, $y(j)$, and $z(j)$ together into $r(3,j)$, $x(j)$, $y(j)$ and $z(j)$ will very likely fall in the same cache line. Then for each iteration, there could still be one cache line fetched, but only one, not three. This will reduce traffic to the cache by a factor of three.

The following two programs, one without grouping and the other with grouping, were compiled with, for example, `f90 -o index_no_group index_no_group.f -lperfix`. The performances of these two programs were monitored using `dtime` and `libperfix`. The events, `e0` and `e1`, being monitored with `libperfix` are chosen by user. Dimension of arrays x , y and z are also from input.

Sample program : `index_no_group.f`

```

program index_no_group

integer e0,e1

read (5,*) nmax
read (5,*) niteration
read (5,*) e0,e1

write (6,*) 'dimension for array = ',nmax
write (6,*) 'number of iteration = ',niteration
write (6,*) "Events monitored : ", e0,e1

call group(nmax,niteration,e0,e1)

stop
end

subroutine group(nmax,niteration,e0,e1)

dimension index(nmax)
dimension x(nmax),y(nmax),z(nmax)
real*4 dtime,tarray(2)

integer*8 c0, c1
integer e0, e1

d = 0.0

call random_number(x)
call random_number(y)
call random_number(z)

do i=1,nmax
  index(i)=int(x(i)*nmax)
end do

call start_counters(e0,e1)

t1=dtime(tarray)

do iteration=1,niteration
do i=1,nmax
  j=index(i)
  d = d + (x(j)*x(j) + y(j)*y(j) + z(j)*z(j))
end do
end do

t2=dtime(tarray)

call read_counters(e0,c0,e1,c1)
call print_counters(e0,c0,e1,c1)
call print_costs(e0,c0,e1,c1)

write (6,*) d
write (6,*) 'time used for do loop = ',t2

return
end

```

Sample program : index_yes_group.f

```
program index_yes_group
integer e0,e1

read (5,*) nmax
read (5,*) niteration
read (5,*) e0,e1

write (6,*) 'dimension for array = ',nmax
write (6,*) 'number of iteration = ',niteration
write (6,*) "Events monitored : ", e0,e1

call group(nmax,niteration,e0,e1)

stop
end

subroutine group(nmax,niteration,e0,e1)
dimension index(nmax)
dimension x(nmax),y(nmax),z(nmax),r(3,nmax)
real*4 dtme,tarray(2)

integer*8 c0, c1
integer e0, e1

d = 0.0

call random_number(x)
call random_number(y)
call random_number(z)

do i=1,nmax
  r(1,i)=x(i)
  r(2,i)=y(i)
  r(3,i)=z(i)
end do

do i=1,nmax
  index(i)=int(x(i)*nmax)
end do

call start_counters(e0,e1)

t1=dtme(tarray)

do iteration=1,niteration
do i=1,nmax
  j=index(i)
!   d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))
  d = d + (r(1,j)*r(1,j)+r(2,j)*r(2,j)+r(3,j)*r(3,j))
end do
end do

t2=dtme(tarray)

call read_counters(e0,c0,e1,c1)
call print_counters(e0,c0,e1,c1)
call print_costs(e0,c0,e1,c1)

write (6,*) d
write (6,*) 'time used for do loop = ',t2

return
end
```

Performances of the above two programs are compared for two different dimensions, 100,000 and 1,000,000. The events monitored are event 0, event 23 in one run and event 0 and event 26 in another run.

Performance of index_no_group.f on lomax (R12000, 400MHz)

niteration = 10 in both runs	nmax=100,000	nmax=1,000,000
dtme in first run	0.156475	9.46948814
0 Cycles Counts	62,585,033	2,511,033,495
0 Cycles Typical Time	0.156463	6.277584
23 TLB misses Counts	250	26,994,065
23 TLB misses Typical Time	0.000049	5.248996
dtme in second run	0.156079993	9.30582333
0 Cycles Counts	62,447,728	2,446,463,929
0 Cycles Typical Time	0.156119	6.116160
26 Secondary Data Cache Misses Counts	12,719	9,629,030
26 Secondary Data Cache Misses Typical Time	0.003176	2.404610

Performance of index_yes_group.f (lomax, R12000, 400MHz)

niteration = 10 in both runs	nmax=100,000	nmax=1,000,000
dtime in first run	0.198895007	4.63336992
0 Cycles Counts	79,521,944	1,423,954,107
0 Cycles Typical Time	0.198805	3.559885
23 TLB misses Counts	670	9,000,009
23 TLB misses Typical Time	0.000130	1.750052
dtime in second run	0.201848999	4.6317358
0 Cycles Counts	80,738,797	1,423,172,151
0 Cycles Typical Time	0.201847	3.557930
26 Secondary Data Cache Misses Counts	12,934	4,007,761
26 Secondary Data Cache Misses Typical Time	0.003230	1.000838

Quiz : Why wasn't there a performance improvement using nmax=100,000 when the second program was used compared to when the first program was used ?

Quiz : If in the above two programs, array accessing is not through an index(i), but through i itself, and thus a stride one access is achieved, will the second program with grouping perform better than the first program without grouping ? Try nmax=1,000,000 and nmax=1024*1024 and see the difference.

● **Remove Cache Trashing by Re-dimensioning Array Sizes not to be Power of Two**

If you have tried answering the above quiz, you probably would have realized that when the dimension of x, y and z arrays is 1024*1024 (=4MB), a phenomenon called cache trashing occurs when x(i), y(i) and z(i) are accessed at the same time. By grouping the three arrays into one, the performance is improved tremendously. Thus **grouping is a good method to eliminate cache trashing when applicable**.

In the following, another technique is introduced for eliminating cache trashing.

Arrays with sizes to be powers of two as x, y and z in index_no_group.f (with nmax=1024*1024) and a, b, c, and d arrays shown in L2_cache_trash.f are more likely to have a(1), b(1), c(1) and d(1) having the same middle address bits and thus mapped to the same cache line. By re-dimensioning these arrays allows them to map to different cache lines and thus avoid cache trashing.

Sample program : L2_cache_trash_dimen.f

```
program L2_cache_trash_dimen
! Redimension arrays so that their sizes are not power of two.
! Accessing a(i,j), b(i,j), c(i,j), and d(i,j) simultaneously
! will not cause cache trashing because they do not use the
! same cache line anymore.
  dimension a(1024+1,1024), b(1024+1,1024), c(1024+1,1024),
    @      d(1024+1,1024)
  call random_number(b)
  call random_number(c)
  call random_number(d)

  do j=1,1024
  do i=1,1024
    a(i,j)=b(i,j)+c(i,j)*d(i,j)
  end do
  end do

  write (12) a

  stop
end
```

%f90 -o L2_cache_trash_dimen L2_cache_trash_dimen.f **! default optimization -O0**

Performance is improved compared to [L2_cache_trash.f](#) as evidenced from perfex -a -x -y output

```
/bin/time perfix -a -x -y L2_cache_trash_dimen
WARNING: Multiplexing events to project totals--inaccuracy possible.
Summary for execution of L2_cache_trash_dimen

Based on 250 MHz IP27
MIPS R10000 CPU
CPU revision 3.x

Event Counter Name                                Counter Value    Typical
                                                    Time (sec)       Minimum
                                                    Time (sec)       Maximum
                                                    Time (sec)
=====
0 Cycles.....                                134959056         0.539836         0.539836         0.539836
16 Cycles.....                                134959056         0.539836         0.539836         0.539836
14 ALU/FPU progress cycles.....              80417520          0.321670         0.321670         0.321670
2 Issued loads.....                          31936880          0.127748         0.127748         0.127748
18 Graduated loads.....                      31885184          0.127541         0.127541         0.127541
3 Issued stores.....                         9761088           0.039044         0.039044         0.039044
19 Graduated stores.....                     9755152           0.039021         0.039021         0.039021
21 Graduated floating point instructions..... 7863872           0.031455         0.015728         1.635685
26 Secondary data cache misses.....           81168             0.024513         0.016026         0.027272
6 Decoded branches.....                     4924992           0.019700         0.019700         0.019700
7 Quadwords written back from scache.....     534784            0.013690         0.009049         0.013690
25 Primary data cache misses.....            361888            0.013042         0.004082         0.013042
22 Quadwords written back from primary data cache..... 597184            0.009197         0.007501         0.010630
9 Primary instruction cache misses.....       11376             0.000820         0.000256         0.000820
23 TLB misses.....                          1360              0.000370         0.000370         0.000370
24 Mispredicted branches.....                28064             0.000159         0.000072         0.000586
10 Secondary instruction cache misses.....     128              0.000039         0.000025         0.000043
31 Store/prefetch exclusive to shared block in scache..... 16              0.000000         0.000000         0.000000
1 Issued instructions.....                  142623200         0.000000         0.000000         0.570493
4 Issued store conditionals.....              0                0.000000         0.000000         0.000000
5 Failed store conditionals.....              0                0.000000         0.000000         0.000000
8 Correctable scache data array ECC errors..... 0                0.000000         0.000000         0.000000
11 Instruction misprediction from scache way prediction table.. 624              0.000000         0.000000         0.000002
12 External interventions.....               432              0.000000         0.000000         0.000000
13 External invalidations.....               1856             0.000000         0.000000         0.000000
15 Graduated instructions.....               15983552          0.000000         0.000000         0.639334
17 Graduated instructions.....               160007632         0.000000         0.000000         0.640031
20 Graduated store conditionals.....           0                0.000000         0.000000         0.000000
27 Data misprediction from scache way prediction table..... 3488             0.000000         0.000000         0.000014
28 External intervention hits in scache.....   416              0.000000         0.000000         0.000000
29 External invalidation hits in scache.....   304              0.000000         0.000000         0.000000
30 Store/prefetch exclusive to clean block in scache.....    0                0.000000         0.000000         0.000000

Statistics
=====
Graduated instructions/cycle.....            1.184311
Graduated floating point instructions/cycle..... 0.058269
Graduated loads & stores/cycle.....           0.308541
Graduated loads & stores/floating point instruction..... 5.295144
Mispredicted branches/Decoded branches..... 0.005698
Graduated loads/Issued loads.....             0.998381
Graduated stores/Issued stores.....            0.999392
Data mispredict/Data scache hits.....         0.012425
Instruction mispredict/Instruction scache hits..... 0.055477
L1 Cache Line Reuse.....                     114.064152
L2 Cache Line Reuse.....                      3.458506
L1 Data Cache Hit Rate.....                   0.991309
L2 Data Cache Hit Rate.....                   0.775710
Time accessing memory/Total time.....         0.378794
Time not making progress (probably waiting on memory) / Total time..... 0.404134
L1-L2 bandwidth used (MB/s, average per process)..... 39.151430
Memory bandwidth used (MB/s, average per process)..... 35.095918
MFLOPS (average per process).....             14.567144

real 1.317
user 0.601
sys 0.349
```

Compiler Assisted Optimization

Profiling reveals which code to tune. The most important tool for tuning is the compiler. The Silicon Graphics compilers are flexible and offer a wide variety of compiler options to control their operation. Refer to the SGI documents listed below to find details about the compiler.

- [MIPSpro 7 Fortran 90 Commands and Directives Reference Manual](#)
- [Chapter 5. Using Basic Compiler Optimization](#) in SGI's Origin2000 and Onyx2 Performance Tuning and Optimization Guide.
- [MIPS Compiling and Performance Tuning Guide](#)
- [MIPSpro 64-Bit Porting and Transition Guide](#)

The default version of compiler used on tuning, hopper, steger and lomax is **MIPSpro.7.3.1.1m**. Use the **module** commands If you need a different version.

Useful Files Generated by Compiler

By default, several files are created during processing. **Some of these files are useful for finding out what the compiler has done for your code.** The compiler adds a suffix to the file portion of the file name and write the files it creates to your working directory.

Files	Content
file I	Assembler listing file. To retain this, specify the -LIST option.
file L	Listing file containing a cross reference and a source listing. To retain this file, specify the -listing option.
file s	Assembly language file. To retain this file, specify the -S or -keep option.
file.w2 f	Fortran transformation file. To retain this file, specify -FLIST:=ON
file.w2 c.c	C transformation file. To retain this file, specify -CLIST:=ON Note: This option does not work with C++.
file list	APO listing file. To retain this, specify the -apolist option.

- Note:**
- Use the assmbleer listing file (file.I) to find out the default compiler options settings at -O0, -O1, -O2, -O3 and -Ofast for option groups:-DEBUG; -LANG; -LIST; -OPT; -LNO; -TARG; -TENV; -FLIST; -CLIST; etc.
 - Use the assembly language file (file.s) to find out the scheduled instructions of your code. If -O3 or -Ofast is used, this file provides information of schedules done by software pipelining.
 - Use the transformation file (file.w2f, file.w2c.c) to find out what transformations are performed by LNO, IPA and other components of the compiler.
 - If you use an older version of compiler, for example, MIPSpro.7.2.1.1m, specifying -apolist will create a file.I file which provides the same information as a file.list generated by newer versions, such as MIPSpro.7.3.1.1m of compiler.

Optimization Option Groups:

Options Group	Function
-On	Basic Optimization
-OPT:	Miscellaneous Optimization Specification
-SWP:	Software Pipelining Specification
-LNO:	Loop Nest Optimization Specification
-IPA:	Inter-Procedural Analysis Specification
-INLINE:	Standalone Inliner Specification
-TENV:	Target Environment Specification
-TARG:	Target Architecture/platform Specification

• Basic Optimization Levels

The basic optimization flag is -On, where n is 0, 1, 2, 3, or fast. This flag controls which optimizations the compiler will attempt: the higher the optimization level, the more aggressive the optimizations that will be tried. In general, the higher the optimization level, the longer compilation takes.

Options	Action	Notes
O0	No optimization	Default
O1	Local optimization	.
O2	Extensive optimization	Optimizations performed at this level are almost always beneficial. No software pipelining and loop nest optimization.
O3	Aggressive optimization	May generate results that differ from those obtained when -O2 is specified. Enables software pipelining and loop nest optimizations.
Ofast	Maximizes performance for the target platform ipxx processor type	The order of operations may be different from that described in the Fortran standard. The ordering is different because -OPT:roundoff=3 is put into effect when -Ofast is specified. Floating-point accuracy may be affected. Enables software pipelining, loop nest optimizations, interprocedural analysis and arithmetic rearrangements.

Suggestions (Quoted from [Chapter 5, Using Basic Compiler Optimization](#) in SGI's Origin2000 and Onyx2 Performance Tuning and Optimization Guide)

o Use -O0 for Debugging

Use the lowest optimization level, -O0, when debugging your program. This flag turns off all optimizations, so there is a direct correspondence between the original source code and the machine instructions the compiler generates.

You can run a program compiled to higher levels of optimization under a symbolic debugger, but the debugger will have trouble showing you the program's progress statement by statement, because the compiler often merges statements and moves code around. An optimized program, under a debugger, can be made to stop on procedure entry, but cannot reliably be made to stop on a specified statement.

-O0 is the default when you don't specify an optimization level, so be sure to specify the optimization level you want.

o Start with -O2 for All Modules

A good beginning point for program tuning is optimization level -O2 (or, equivalently, -O). This level performs extensive optimizations that are conservative; that is, they will not cause the program to have numeric roundoff characteristics different from an unoptimized program. Sophisticated (and time-consuming) optimizations such as software pipelining and loop nest optimizations are not performed.

In addition, the compiler does not rearrange the sequence of code very much at this level. At higher levels, the compiler can rearrange code enough that the correspondence between a source line and the generated code becomes hazy. If you profile a program compiled at the -O2 level, you can make use of a prof -heavy report (see "Including Line-Level Detail"). When you compile at higher levels, it can be difficult to interpret a profile because of code motion and inlining.

Use the -O2 version of your program as the baseline for performance. For some programs, this may be the fastest version. The higher optimization levels have their greatest effects on loop-intensive code and math-intensive code. They may bring little or no improvement to programs that are strongly oriented to integer logic and file I/O.

o Compile -O3 or -Ofast for Critical Modules

You should identify the program modules that consume a significant fraction of the program's run time and compile them with the highest level of optimization. This may be specified with either -O3 or -Ofast. Both flags enable software pipelining and loop nest optimizations. The difference between the two is that -Ofast includes additional optimizations:

- Interprocedural analysis
- Arithmetic rearrangements that can cause numeric roundoff to be different from an unoptimized program
- Assumption that certain pointer variables are independent, not aliased

With no argument, -Ofast assumes the default target for execution (see "Compiler Defaults"). You can specify which machine will run the generated code by naming the "ip" number of the CPU board. The complete list of valid board numbers is given in the cc(1) reference page. The ip number of any system is displayed by the command hinv -c processor. For all SNO systems, use -Ofast=ip27 (this implies the flags -r10000,-TARG:proc=r10000 and -TARG:platform=ip27).

Example : Compiler Assisted Optimization

Performance (from time command) of [L2_cache_trash.f](#) with various basic optimization levels

time	O0	O1	O2	O3	Ofast
real	2.932	2.968	1.628	0.630	0.619
user	2.684	2.728	1.411	0.411	0.411
sys	0.196	0.190	0.192	0.196	0.186

Performance of this program is best when -O3 (or -Ofast) is used. The most important optimization for this program by -O3 is the **local array padding** (a suboption of Loop Nest Optimization) which removes the cache_trashing problems. This is evidenced in the transformed code shown below:

The arrays declared in [L2_cache_trash.f](#) are:

dimension a(1024,1024), b(1024,1024), c(1024,1024), d(1024,1024)

LNO transformed (as seen in `L2_cache_trash.w2f.f` obtained with `-O3 -FLIST:=ON`) these arrays as:

```
REAL(4) A(1024_8, 1024_8)
REAL(4) B(1050_8, 1024_8)
REAL(4) C(1050_8, 1024_8)
REAL(4) D(1050_8, 1024_8)
```

The best performance from -O3 is also reflected in the perfex -a -x -y output (shown below) as compared to (i) [the perfex output with -O0](#) for the same program and (ii) [the perfex output with -O0 and manual padding](#) as shown in [L2 cache trash dimen.f](#)

```

/bin/time perf -a -x -y L2_cache_trash_03
WARNING: Multiplexing events to project totals--inaccuracy possible.
Summary for execution of L2_cache_trash_03

Based on 250 MHz IP27
MIPS R10000 CPU
CPU revision 3.x

Event Counter Name
Counter Value
Typical Time (sec)
Minimum Time (sec)
Maximum Time (sec)

0 Cycles..... 93268624 0.373074 0.373074 0.373074
16 Cycles..... 93268624 0.373074 0.373074 0.373074
14 ALU/FPU progress cycles..... 26248416 0.104994 0.104994 0.104994
2 Issued loads..... 24451136 0.097805 0.097805 0.097805
18 Graduated loads..... 23960624 0.095842 0.095842 0.095842
21 Graduated floating point instructions..... 9054560 0.036218 0.036218 0.036218
3 Issued stores..... 8897264 0.035589 0.035589 0.035589
19 Graduated stores..... 8887248 0.035549 0.035549 0.035549
6 Decoded branches..... 3784208 0.015137 0.015137 0.015137
22 Quadwords written back from primary data cache..... 404000 0.006222 0.005074 0.007191
25 Primary data cache misses..... 20880 0.000753 0.000236 0.000753
9 Primary instruction cache misses..... 9744 0.000702 0.000219 0.000702
7 Quadwords written back from scache..... 17024 0.000436 0.000288 0.000436
23 TLB misses..... 608 0.000166 0.000166 0.000166
26 Secondary data cache misses..... 160 0.000048 0.000032 0.000054
24 Mispredicted branches..... 5200 0.000030 0.000013 0.000109
10 Secondary instruction cache misses..... 96 0.000029 0.000019 0.000032
31 Store/prefetch exclusive to shared block in scache..... 16 0.000000 0.000000 0.000000
1 Issued instructions..... 55238128 0.000000 0.000000 0.220953
4 Issued store conditionals..... 0 0.000000 0.000000 0.000000
5 Failed store conditionals..... 0 0.000000 0.000000 0.000000
8 Convertible scache data array ECC errors..... 0 0.000000 0.000000 0.000000
11 Instruction misprediction from scache way prediction table.. 576 0.000000 0.000000 0.000002
12 External interventions..... 288 0.000000 0.000000 0.000000
13 External invalidations..... 4688 0.000000 0.000000 0.000000
15 Graduated instructions..... 62291728 0.000000 0.000000 0.249167
17 Graduated instructions..... 62331088 0.000000 0.000000 0.249324
20 Graduated store conditionals..... 0 0.000000 0.000000 0.000000
27 Data misprediction from scache way prediction table..... 480 0.000000 0.000000 0.000002
28 External intervention hits in scache..... 288 0.000000 0.000000 0.000000
29 External invalidation hits in scache..... 352 0.000000 0.000000 0.000000
30 Store/prefetch exclusive to clean block in scache..... 0 0.000000 0.000000 0.000000

Statistics
=====
Graduated instructions/cycle..... 0.667874
Graduated floating point instructions/cycle..... 0.097080
Graduated loads & stores/cycle..... 0.352186
Graduated loads & stores/floating point instruction..... 3.627771
Mispredicted branches/Decoded branches..... 0.001374
Graduated loads/Issued loads..... 0.979939
Graduated stores/Issued stores..... 0.998874
Data mispredict/Data scache hits..... 0.023166
Instruction mispredict/Instruction scache hits..... 0.059701
L1 Cache Line Reuse..... 1572.173946
L2 Cache Line Reuse..... 129.500000
L1 Data Cache Hit Rate..... 0.999364
L2 Data Cache Hit Rate..... 0.992337
Time accessing memory/Total time..... 0.354772
Time not making progress (probably waiting on memory) / Total time..... 0.718572
L1-L2 bandwidth used (MB/s, average per process)..... 19.117254
Memory bandwidth used (MB/s, average per process)..... 0.785001
MFLOPS (average per process)..... 24.270113

real 1.152
user 0.438
sys 0.372

```

- **Software Pipelining**

Software pipelining is a compiler technique in which the compiler generates sequences of instructions (**for innermost loops**) that are carefully tailored to take maximal advantage of the multiple execution units of a superscalar CPU. [Chapter 5. Using Basic Compiler Optimization](#) in SG's Origin2000 and Onyx2 Performance Tuning and Optimization Guide provides more description regarding software pipelining.

- o The R10000/R12000 CPU can execute up to four instructions per cycle. Superscalar time slots may be filled by any combination of:
 - One load or store operation
 - One ALU 1 instruction
 - One ALU 2 instruction
 - One floating-point add
 - One floating-point multiply

Example: Software Pipelining - Instruction Scheduling

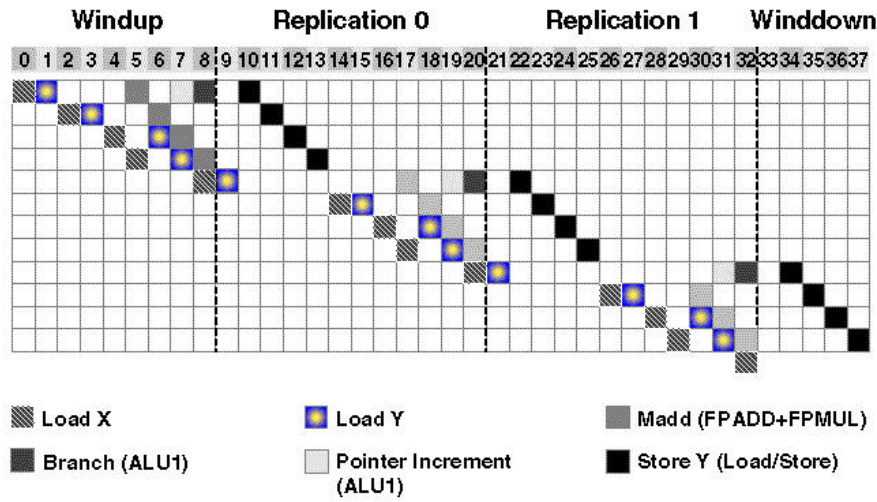
```
do i = 1, n
    y(i) = y(i) + a*x(i)
enddo
```

Each iteration of the loop requires the following instructions:

- Two loads (of $x(i)$ and $y(i)$)
- One store (of $y(i)$)
- One multiply-add
- Two address increments
- One loop-end test
- One branch

The following shows one possible schedule generated by the compiler. Time, in cycles, runs horizontally across the diagram. (obtained from [Chapter 5. Using Basic Compiler Optimization](#) in SGI's Origin2000 and Onyx2 Performance Tuning and Optimization Guide)

In this schedule, the compiler decides to unroll the loop 4 times. Thus, address increment instruction and branch instruction only need to be performed every 4 iterations.



- Software Pipelining can not be done if
 - There are subroutine or function calls in loop : use -INLINE to achieve SWP
 - There are conditionals or branching in the loop (ex: go to statement)
- Software Pipelining is automatically turned on at -O3 and -Ofast.
- Software Pipelining is not done by default at -O0, -O1 and -O2, because it increases compilation time.
- Use -OPT:swp=ON or -SWP:=ON to turn software pipelining on for lower levels of optimization. (not recommended)
- The schedule found can vary depending on subtle code differences or the choice of compiler flags.
- Software pipelining may not always generate an optimal schedule.
- To find how effectively the CPU's hardware resources are being used in the schedule, use -S compiler option to generate an assembly language file (file.s) in which software pipelining messages are included.

Example: Obtain Software Pipelining Messages

%f90 -O3 -S -c [L2_cache_trash.f](#) <- Generates L2_cache_trash.s

%fgrep swps L2_cache_trash.s <- Extracts software pipelining info

```
#<swps>
#<swps> Pipelined loop line 14 steady state <- this line number can be approximate
#<swps>
#<swps>      256 iterations before pipelining
#<swps>      4 unrollings before pipelining <- this loop was unrolled 4 times and then software pipelined
#<swps>      20 cycles per 4 iterations <- Every 4 iterations (1 replication) will be completed in 20 cycles
#<swps>      8 flops      ( 20% of peak) (madds count as 2) <- This loop can achieve 20% of floating-point peak
#<swps>      4 flops      ( 10% of peak) (madds count as 1)
#<swps>      4 madds      ( 20% of peak)
#<swps>      16 mem refs   ( 80% of peak) <- Each replication contains 16 memory reference
#<swps>      5 integer ops ( 12% of peak)
#<swps>      25 instructions ( 31% of peak)
#<swps>      2 short trip threshold
#<swps>      13 integer registers used.
#<swps>      18 float registers used.
#<swps>
```

%fgrep swpf L2_cache_trash.s <- Extracts software pipelining failure

```
#<swpf> Loop line 10 wasn't pipelined because:
#<swpf>      Function call in the loop body
#<swpf>
#<swpf> Loop line 11 wasn't pipelined because:
#<swpf>      Function call in the loop body
#<swpf>
#<swpf> Loop line 12 wasn't pipelined because:
#<swpf>      Function call in the loop body
#<swpf>
```

● Loop Nest Optimization (LNO)

Numerical programs often spend most of their time executing loops. Loop nest optimization is a compiler optimization technique in which the statements in nested loops are transformed to give better cache use or better instruction scheduling.

- Major optimization performed by LNO:
 - [Array Padding](#)
 - [Loop Interchange](#)
 - [Outer Loop Unrolling](#)
 - [Cache Blocking](#)
 - [Loop Fusion](#)
 - [Loop Fission](#)
 - [Prefetching](#)
 - [Gather-Scatter](#)
 - [Vector Intrinsics](#)
- LNO is used by default at -O3 and -Ofast.
- Use -LNO:opt=0 if you want to turn off LNO optimization at -O3 and -Ofast.
- Most optimization can be individually controlled using LNO suboptions, directives and pragma.
- LNO: option group is enabled only if -O3 is also specified on the compiler command line.
- View the transformation LNO perform for your code in file.w2f.f or file.w2c.c

```
f90 -O3 -FLIST:=ON file.f
cc -O3 -CLIST:=ON file.c
```

o Array Padding

- An unlucky alignment of arrays can cause significant performance penalties from cache trashing. LNO automatically pads arrays, by spacing them out, to eliminate or reduce cache trashing.
- In the [Compiler Assisted Optimization](#) example for [L2_cache_trash.f](#) shown above, LNO automatically performs padding by increasing the first dimension of arrays as shown explicitly in the transformed code.

dimension a(1024,1024), b(1024,1024), c(1024,1024), d(1024,1024) ->

```
REAL(4) A(1024_8, 1024_8)
REAL(4) B(1050_8, 1024_8)
REAL(4) C(1050_8, 1024_8)
REAL(4) D(1050_8, 1024_8)
```

- Padding performed by LNO is not always shown explicitly in file.w2f.f or file.s.

Sherry, think about array padding for common blocks

o Loop Interchange

- In nested loops, a few factors need to be considered when deciding on the order of loops.
 - cache misses - stride-1 memory reference reduces cache misses when loop dimensions are sufficiently large.

In [loop_interchange.f](#), reorder loops from

```
do i=1,1024
do j=1,100
do k=1,100
-->
k=1,100
do j=1,100
do i=1,1024
```

allow stride-1 memory reference and greatly reduce both tlb misses and secondary data cache misses.

- instruction scheduling and loop overhead -

```
dimension a(2,100)
do i=1,2
do j=1,100
a(i,j)=0.0
enddo
enddo
-->
dimension a(2,100)
do j=1,100
do i=1,2
a(i,j)=0.0
enddo
enddo
```

In this case, the a(2,100) array fits in cache, and with the original loop order (i->j), the code achieve full cache reuse. With the new loop order (j->i), the shorter loop i is inside and software pipelining loop i causes more loop overhead.

- LNO considers these factors when deciding on whether it should reorder loops.
- -LNO:interchange=ON by default with -O3
- If you know that LNO has made the wrong loop interchange, you can instruct it to make a correction by:
 - turn off loop interchange
 - for the entire module
 - LNO:interchange=OFF
 - for a single loop nest, use directive or pragma:


```
C*$* no interchange      for Fortran codes
#pragma no interchange    for C codes
```
 - tell the compiler which order you want the loops in


```
C*$* interchange (i,j,k)    for Fortran codes
#pragma interchange (i,j,k) for C codes
```

this order requests that i is outermost, and k is innermost.

o Outer Loop Unrolling

- With loop unrolling, multiple copies of the loop body are generated.
- Inner loop unrolling, when beneficial, occurs automatically when -O2 or -O3 or software pipelining is in effect.

```
do j=1,10
do i=1,100
a(i,j)=b(i,j) + 1
end do
end do
-->
do j=1,10
do i=1,100,2
a(i,j)=b(i,j) + 1
a(i+1,j)=b(i+1,j) + 1
end do
end do
```

- Outer loop unrolling occurs automatically when -O3 (-LNO) is in effect.

```
do j=1,10
do i=1,100
a(i,j)=b(i,j) + 1
end do
end do
-->
do j=1,10,2
do i=1,100
a(i,j)=b(i,j) + 1
a(i,j)=b(i,j+1) + 1
end do
end do
```

- Outer loop unrolling applies to (i) outer-most loop unrolling, (ii) middle loop unrolling and (iii) outer-most loop unrolling + middle loop unrolling

In the matrix multiplication example below, the outer-most loop j is unrolled 2 times and the middle k loop is unrolled 4 times by LNO if you compile with -O3 -LNO:blocking=off. [Cache Blocking](#) is turned off here for the purpose of demonstrating the effect of loop unrolling without the complication from cache blocking.

```

parameter (m=1200,n=1200,l=1200)
dimension a(m,l),b(l,n),c(m,n)

do j = 1, n
do k = 1, l
do i = 1, m
c(i,j) = c(i,j) + a(i,k)*b(k,j)
enddo
enddo
enddo

->
do j = 1, 1200, 2
do k = 1, 1200, 4
do i = 1, 1200
c(i,j) = c(i,j) + a(i,k)*b(k,j)
c(i,j) = c(i,j) + a(i,k+1)*b(k+1,j)
c(i,j) = c(i,j) + a(i,k+2)*b(k+2,j)
c(i,j) = c(i,j) + a(i,k+3)*b(k+3,j)
c(i,j+1) = c(i,j+1) + a(i,k)*b(k,j+1)
c(i,j+1) = c(i,j+1) + a(i,k+1)*b(k+1,j+1)
c(i,j+1) = c(i,j+1) + a(i,k+2)*b(k+2,j+1)
c(i,j+1) = c(i,j+1) + a(i,k+3)*b(k+3,j+1)
enddo
enddo
enddo

```

- The numbers of unrolling will be determined by the compiler for best performance.
- Loop unrolling should be indicated in the assembly listing file, file.s but may not be shown explicitly in file.w2f.f.
- One can control or fine-tune loop unrolling
 - for all loops:
 - -LNO:outer_unroll=n
unrolls every outer loop for which unrolling is possible by exactly n times
 - -LNO:outer_unroll_max=n
tells the compiler that it may unroll any outer loop by at most n times
 - -LNO:outer_unroll_prod_max=n
indicates that the product of unrolling the various outer loops in a given loop nest will not exceed n times
 - for individual loop:


```

C*$* unroll (n)      for Fortran codes
#pragma unroll (n)   for C codes

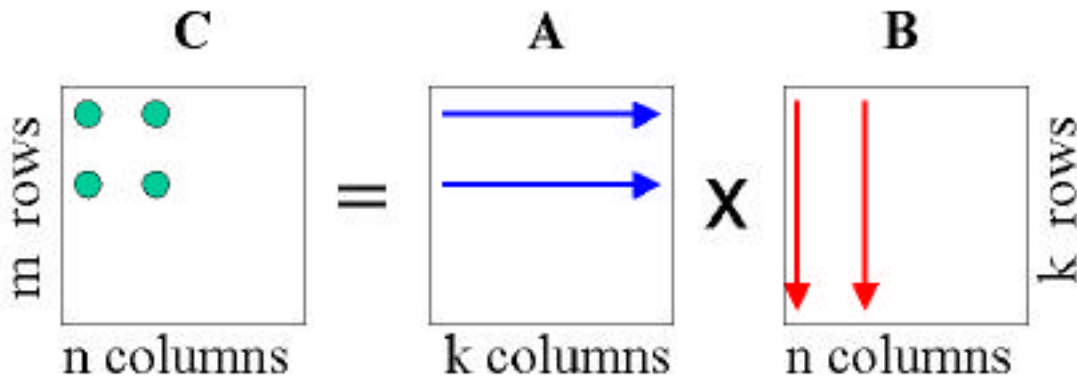
n=0  default unrolling is applied
n=1  no unrolling
n>1  loop unrolls n times
          
```

- Loop unrolling may not be performed if the compiler determines that unrolling is not safe or beneficial.

o Cache Blocking

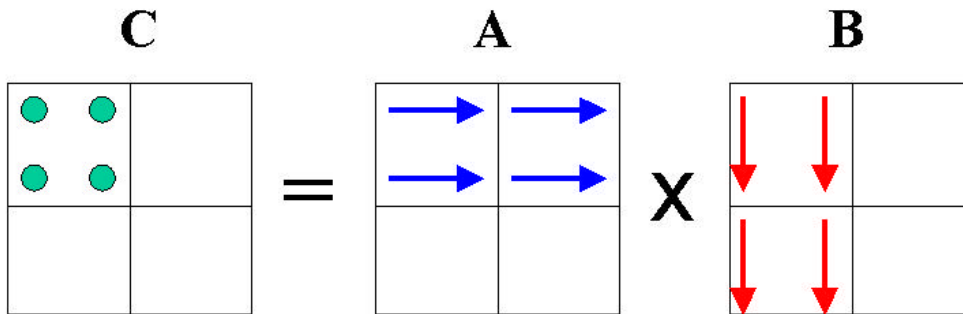
- When data structures are too big to fit in the cache, cache misses are more likely to occur. Cache blocking is a technique to break up data structures into smaller pieces that will fit in the cache, thus reducing the probability of cache misses and main memory accesses.

Example : Improve Cache Utilization with Cache Blocking



In matrix multiplication, calculating one element of C requires reading an entire row of A and an entire column of B . Calculating an entire column of C (m elements) requires reading all rows (m rows) of A once and re-reading one column of B m times. Thus, Calculating n columns of C requires reading all rows of A n times and all columns of B m times.

If A and B do not fit in the cache, the earlier rows (of A) and columns (of B) (or some elements of them) are likely to be displaced by the later ones. Thus, when the earlier rows and columns are needed again, they have to be re-loaded from memory.



In blocking, matrix multiplication of the original arrays is transformed into matrix multiplication of blocks. For example,

```
C_block(1,1)=A_block(1,1)*B_block(1,1) + A_block(1,2)*B_block(2,1)
```

When three blocks (one from C, one of A and one from B) all fit in the cache, the elements of these blocks need to be read in from memory only once for each block multiplication. **The number of memory accesses is reduced from the number of rows or columns to the number of blocks.**

Sample program : matrix.f

```
program matrix_mult
parameter (m=1200,n=1200,l=1200)
dimension a(m,l),b(l,n),c(m,n)

call random_number(a)
call random_number(b)

do j = 1, n
do k = 1, l
do i = 1, m
c(i,j) = c(i,j) + a(i,k)*b(k,j)
enddo
enddo
enddo

write (12) c

stop
end
```

Matrix multiplication of original arrays transformed into a series of block multiplication

```
DO tile2J = 1, 1200, 240           ! 5 blocks along j direction
DO tile2K = 1, 1200, 300           ! 4 blocks along k direction
DO I = 1, 1200, 5                  ! unroll i 5 times
DO J = tile2J, (tile2J + 239), 3   ! unroll j in each block 3 times
DO K = tile2K, (tile2K + 299), 1
C(I, J) = (C(I, J) + (A(I, K) * B(K, J)))
C(I + 1, J) = (C(I + 1, J) + (A(I + 1, K) * B(K, J)))
C(I + 2, J) = (C(I + 2, J) + (A(I + 2, K) * B(K, J)))
C(I + 3, J) = (C(I + 3, J) + (A(I + 3, K) * B(K, J)))
C(I + 4, J) = (C(I + 4, J) + (A(I + 4, K) * B(K, J)))
C(I, J + 1) = (C(I, J + 1) + (A(I, K) * B(K, J + 1)))
C(I + 1, J + 1) = (C(I + 1, J + 1) + (A(I + 1, K) * B(K, J + 1)))
C(I + 2, J + 1) = (C(I + 2, J + 1) + (A(I + 2, K) * B(K, J + 1)))
C(I + 3, J + 1) = (C(I + 3, J + 1) + (A(I + 3, K) * B(K, J + 1)))
C(I + 4, J + 1) = (C(I + 4, J + 1) + (A(I + 4, K) * B(K, J + 1)))
C(I, J + 2) = (C(I, J + 2) + (A(I, K) * B(K, J + 2)))
C(I + 1, J + 2) = (C(I + 1, J + 2) + (A(I + 1, K) * B(K, J + 2)))
C(I + 2, J + 2) = (C(I + 2, J + 2) + (A(I + 2, K) * B(K, J + 2)))
C(I + 3, J + 2) = (C(I + 3, J + 2) + (A(I + 3, K) * B(K, J + 2)))
C(I + 4, J + 2) = (C(I + 4, J + 2) + (A(I + 4, K) * B(K, J + 2)))
END DO
END DO
END DO
END DO
END DO
```

- -LNO:blocking=ON by default with -O3
- Cache blocking, loop unrolling, loop interchange and array padding are considered together by the compiler to achieve optimum cache utilization.
- The performance of the blocked algorithm matches the performance of the unblocked algorithm on a size that fits entirely on the L2 cache. For example, if the L2 cache size is 4MB as in our R10000 Origins Hopper and Steger, all three arrays A, B, C together fit in L2 cache when the dimensions $m=n=l \sim 600$.

As the array sizes get larger, the performance of matrix multiplication (such as usertime, MFLOPS, and L2 cache hit rate) gets worse when no cache blocking is used.

The table here shows the performance of matrix.f with and without cache blocking for various array sizes (with $m=n=l$). The program is compiled with

```
f90 -O3           to enable cache blocking
f90 -O3 -LNO:blocking=off to disable cache blocking
```

array dimension	cache blocking	user time	MFLOPS	L2 cache hit rate
600	yes	1.062	229.8	0.996
600	no	1.021	239.8	0.998
800	yes	2.454	221.6	0.982
800	no	2.769	200.2	0.995
1000	yes	4.685	221.8	0.991
1000	no	5.164	193.3	0.973
1200	yes	8.098	215.7	0.977
1200	no	9.970	175.8	0.891
2400	yes	64.498	214.5	0.980
2400	no	104.780	132.2	0.837
4800	yes	526.705	210.3	0.989
4800	no	906.898	122.1	0.838

- One can control or fine-tune cache blocking
 - to specify block size for L1 cache, L2 cache or both

- for the whole module

-LNO:blocking_size={l1},{l2}

- for a single loop nest

```
c*$* BLOCKING SIZE {l1},{l2}
$pragma blocking size ({l1},{l2})
```

- to turn off cache blocking

- for the whole module

-LNO:blocking=off

- for a single loop nest

```
c*$* NO BLOCKING
$pragma no blocking
```

o Loop Fusion

- Loop fusion combines two or multiple loops together.
- It is a technique that can improve cache performance and enable other optimizations, such as loop interchange and cache blocking.
- When necessary, loop peeling is performed prior to loop fusion.

```

->
do i=1,n
a(i)=b(i+1)+b(i-1)      a(1)=b(2)+b(0)    ! loop peeling
enddo

do i=1,n
b(i)=a(i+1)+a(i-1)
enddo

do i=2,n
a(i)=b(i+1)+b(i-1)      ! loop fusion
b(i-1)=a(i)+a(i-2)
enddo

b(n)=a(n+1)+a(n-1)      ! loop peeling
```

With the original program, the two loops can not be fused together because of the dependency of a(i) and a(i+1). If n is sufficiently large, in each loop you need to bring the entire a and b matrices into the cache.

By peeling off the calculation of a(1) and b(n), the program can be rewritten to avoid the dependency and the two loops fused together. This eliminates cache misses and memory references.

- The potential drawback with loop fusion :

With larger loop body, it places more demands on compiler regarding register allocation and software pipelining. It thus increases the compilation time.

- One can control loop fusion

- for the whole module

```
-LNO:fusion=0    disable fusion
-LNO:fusion=1    default with -O3
-LNO:fusion=2    try fusion before fission
```

If -LNO:fusion=n and -LNO:fission=n are both set to 1 or 2, fusion is performed.

- for individual loops

```
C*$* FUSE        request fusing following loops
#pragma fuse
C*$* NO FUSION   prevent fusing loops
#pragma no fusion
```

o Loop Fission

- Loop fission breaks larger loops into smaller loops. This is to balance the negative effect by loop fusion.
- Loop fission can requires the invention of new variables.

```

->
do i=1,n
s=float(i)
do i=1,n
  stemp(i)=float(i) ! new variable introduced
end do
do i=1,n
  a(i)=stemp(i)
enddo

```

- An example of using loop fission to facilitate loop interchange

```

->
do i=1,m
do j=1,n
  b(i,j)=a(i,j)
enddo
do j=1,n
  c(i,j)=b(i+k,j)
enddo
enddo

do j=1,n
do i=1,m ! loop interchange
  b(i,j)=a(i,j)
enddo
do i=1,m ! loop interchange
  c(i,j)=b(i+k,j)
enddo
enddo

```

- Loop fission is useful for vectorization of intrinsics. LNO attempts to split vectorizable intrinsics into their own loops. If successful, each such loop is collapsed into a single call to the corresponding [vector intrinsic](#).

- One can control loop fission

- for the whole modules

```

-LNO:fission=0    disable fission
-LNO:fission=1    default with -O3
-LNO:fission=2    try fission before fusion

```

- for individual loops

```

C*$* FISSION      request fissioning of a loop
#pragma fission
C*$* NO FISSION    prevent fissioning of a loop
#pragma no fission

```

o Prefetching

- Prefetching moves data from main memory into cache before their use. This allows some or all of the memory latency to be hidden. It is useful in compute-intensive operations where data is too large to fit in the cache.

- LNO estimates which references will be cache misses and takes different prefetching actions.

- L2 misses

Regular prefetching : inserting prefetch instructions in the program with unrolling

```

->
do i=1,n
a = a + b(i)
end do

do i=1,n
  prefetch b(i+16) ! inserted prefetch instruction
  a = a + b(i)
end do

```

You can find the inserted prefetch instructions in file.s but not in file.w2f.f

- L1 misses

Pseudo prefetching : does not insert prefetch instruction, instead, moves loads early in the schedule, exploiting out-of-order execution

```

->
do i=1,4
a = a + b(i)
a = a + b(i+1)
a = a + b(i+2)
a = a + b(i+3)
end do

do i=1,4
  t = b(i+3) ! load b(i+3) in advance
  a = a + b(i)
  a = a + b(i+1)
  a = a + b(i+2)
  a = a + t
end do

```

- Prefetching enabled by default with -O3

- One can control prefetching

```

-LNO:prefetch=0    no prefetches
-LNO:prefetch=1    default with -O3, conservative prefetching
-LNO:prefetch=2    aggressive prefetching

```

o Gather-Scatter

- A do loop that contains a branch (like `if` statement) is not executed efficiently. LNO optimizes such loops with Gather-Scatter.

```

->
do i=1,n
if (t(i).gt.0.0) then
  index=index+1 !gather the indices
  tmp(index)=i !for which condition is true
end if
end do

do i=1,index
  a(tmp(i))=b(tmp(i))
end do
enddo

```

The original code suffers two kinds of efficiency.

- Some array elements are skipped making software pipelining not effective.

- Each time `if` fails, the CPU has already performed a few instructions speculatively before the failure is evident. CPU cycles are wasted on these instructions and on internal resynchronization.

http://www.nasa.gov/~schang/origin_opt.html

- ❑ Gather-Scatter is on by default with -O3
- ❑ One can control Gather-Scatter
 - LNO:gather-scatter=0 disable gather-scatter
 - LNO:gather-scatter=1 for non-nested if statement, default at -O3
 - LNO:gather-scatter=2 for nested if statements, performance may be slower

o Vector Intrinsic

- LNO:vintr=ON by default with -O3
- ❑ LNO can convert some scalar math intrinsic calls (sin, cos, exp, log, sqrt, etc.) into vector calls so that they can take advantage of the vector math routines in the math library, libm. Vector functions are always faster than scalar functions when the vector has at least ten elements.
- ❑ The results of using a vector routine may not agree, bit for bit, with the results of the scalar routine. If it is critical to have numeric agreement precise to the last bit, disable this optimization with -LNO:vintr=OFF.

Example : Use Vector Intrinsic for Better Performance

Sample program : vector_intrinsic.f

```
->
program vector_intrinsic
dimension a(1000,1024),b(1000,1024)
real*4 dtime,tarray(2)

t1=dtime(tarray)
do j=1,1024
do i=1,1000
a(i,j)=float(i+j)
enddo
enddo

a(i,j)=tan(a(i,j))
b(i,j)=exp(a(i,j))
enddo
enddo

t2=dtime(tarray)

write (6,*) 'time spent in loop =',t2
write (12) a
write (12) b

stop
end
```

```
f90 -O3 -o vector_intrinsic vector_intrinsic.f !LNO:vintr=ON by default at -O3
f90 -O3 -LNO:vintr=OFF -o vector_intrinsic_off vector_intrinsic.f ! disable vector instrinsic
```

vintr	t2 from Hopper	t2 from Lomax
ON	0.242 s	0.148 s
OFF	0.383 s	0.240 s

• Inter-Procedural Analysis (IPA)

IPA performs program optimizations that can only be done in the presence of the whole program.

Most compiler optimizations (i.e., without IPA) work within a single procedure (for example, function or subroutine) at a time. This helps keep the problems manageable, and is a key aspect of supporting separate compilation, because it allows the compiler to restrict attention to the current source file.

This intra-procedural focus also presents serious restrictions. By avoiding dependence on information from other procedures, an optimizer is forced to make worst-case assumptions about the possible effects of those procedures. For instance, at boundary points including all procedure calls, the compiler must typically save (and/or restore) the state of all variables to or from memory.

By contrast, Inter-Procedural Analysis (IPA) algorithms analyze more than a single procedure (preferably the entire program) at once. This is done by postponing much of the compilation process until the link step, when all of the program components can be analyzed together.

- o Optimizations performed by MIPSpro compiler's IPA include: (see ipa man page for details)
 - ❑ Inlining
 - ❑ Constant Propagation
 - ❑ Common block array padding - pad the leading dimension to avoid cache conflicts
 - ❑ Dead function elimination
 - ❑ Dead variable elimination
 - ❑ Global name optimizations
 - o -IPA is enabled automatically when -Ofast is used.
 - o Add -IPA for -O2 and -O3 at both the compile and link steps when IPA is desired at these two optimization levels.
 - o IPA is controlled from the command line with two option groups:
 - ❑ -INLINE:...
controls inlining by the standalone inliner. Use when the full IPA is not suitable.
 - ❑ -IPA:...
controls general IPA choices. It also includes a main inliner.
 - o Both IPA and standalone inlining are disabled when -g is specified on the compile line.
- For more information on the individual options in the IPA group, see the ipa man page.

Example : Optimization with Inter-Procedural Analysis

Sample program : ipa.f90 and sub.f90

```

1 program ipa
2 do i=1,1000
3   a=10.0
4   b=1./a
5   write (6,*) a
6   write (6,*) b
7 end do
8
9 call sub(i)
10
11 stop
12 end

1 subroutine sub(i)
2
3 write (6,*) 'i=',i
4 return
5 end

```

% f90 -Ofast -FLIST:=ON -o ipa.exe ipa.f90 sub.f90 ipa.f90: sub.f90: /opt/MIPSPRO/MIPSPRO/usr/lib32/cmplrs/be translates 1.I into 1.w2f.f, based on source 1.I

The transformed code, 1.w2f.f

```

C *****
C Fortran file translated from WHIRL Thu Mar 1 15:29:22 2001
C *****

      PROGRAM MAIN
      IMPLICIT NONE

C
C      **** Variables and functions ****
C
      REAL(4) A
      REAL(4) B
      INTEGER(4) I

C
C      **** statements ****
C
      DO I = 1, 1000, 1
        A = 1.0E+01
        B = 1.0000000149E-01
        WRITE(6, *) A
        WRITE(6, *) B
      END DO
      WRITE(6, *) 'i=', 1001      ! sub has been inlined
      STOP
      END

```

If lines 5 and 6 in ipa.f90 are commented out, the transformed code will be:

```

C *****
C Fortran file translated from WHIRL Thu Mar 1 15:32:50 2001
C *****

      PROGRAM MAIN
      IMPLICIT NONE

C
C      **** statements ****
C
      WRITE(6, *) 'i=', 1001      !dead loop has been eliminated
      STOP
      END

```

• Inlining

Inlining is a compiler optimization technique in which the call to a procedure is replaced by a copy of the body of the procedure, with the actual parameters substituted for the parameter variables.

Benefits of inlining:

- Eliminate the overhead of calling the procedure
- Give compiler more statements to optimize

Disadvantages of inlining:

- Increase compilation time
- Increase size of code

Methods to activate inlining:

- use the main inliner in IPA : automatically activated by default when -Ofast is used. Inlining by the main inliner can be turned off with -IPA:inline=OFF compiler flag. This does not affect the standalone inliner.
- use the standalone inliner in -INLINE : requires the caller and callee programs in the same file. Use -INLINE options to control inlining. If you have included inlining directives in your source code, the -INLINE option must be specified in order for those directives to be honored.

[Chapter 3. General Directives](#) of SGI's [MIPSPRO 7 Fortran 90 COmmands and Directives Reference Manual](#) provides more information about using the INLINE directive.

For more information on the individual options in the INLINE group, see the ipa(5) man page.

Example : Inlining with -INLINE

Sample Program : ipa_sub.f90

```

%cat ipa.f90 sub.f90 > ipa_sub.f90      ! get everything in 1 file in order for -INLINE to work
f90 -O3 -INLINE:must=sub -FLIST:=ON ipa_sub.f90 -> generates ipa_sub.w2f.f, -O3 does not invoke -IPA

```

Transformed code from -INLINE

```

C *****
C Fortran file translated from WHIRL Thu Mar 1 16:47:11 2001
C *****
      PROGRAM MAIN
      IMPLICIT NONE

      **** Variables and functions ****

      REAL(4) A
      REAL(4) B
      INTEGER(4) I

      **** Temporary variables ****

      INTEGER(4) I1

      **** statements ****

      DO I1 = 1, 1000, 1
        A = 1.0E+01
        B = 1.0000000149E-01
        WRITE(6, *) A
        WRITE(6, *) B
      END DO
      I = 1001
      WRITE(6, *) 'i=', I !content of sub has been inlined
      STOP
      END

      SUBROUTINE sub(I)
      IMPLICIT NONE
      INTEGER(4) I

      **** statements ****

      WRITE(6, *) 'i=', I
      RETURN
      END SUBROUTINE

```

Note: You can get the same transformed code if you add the following in the source code just before calling sub in ipa_sub.f90:

```

!$* INLINE HERE (sub)

%f90 -O3 -INLINE -FLIST:=ON ipa_sub.f90

```

• Miscellaneous Compiler Optimizations

The -OPT: option group controls miscellaneous optimizations. This option overrides defaults based on the main optimization level. See opt man page for details.

• Syntax:

```

f90 -OPT:cis=ON:cray_ivdep=OFF b.f
or
f90 -OPT:cis=ON -OPT:cray_ivdep=OFF b.f

```

• Three of the opt subgroups are described here.

```

swp[ = ( ON|OFF ) ]
    swp=ON enables software pipelining. swp=ON is enabled when
    -O3 is in effect. The default is OFF.

```

IEEE_arithmetic=n

This option controls how strictly the generated code adheres to the IEEE 754 standard for floating-point arithmetic. IEEE 754 describes a standard for, among other things, NaN and inf operands, arithmetic round off, and overflow.

n can be one of the following:

n	Description
1	Full compliance with the standard.
2	Inhibits optimizations that produce less accurate results than required by ANSI/IEEE 754-1985. This is the default for -O0, -O1 and -O2.
3	Permits use of the slightly-inexact MIPS IV hardware instructions. (Note: the MIPS IV reciprocal and reciprocal square root do not meet the accuracy specified by the IEEE 754 standard)

Allows compiler optimizations that can produce less accurate inexact results (but accurate exact results) on the target hardware. For example, -OPT:recip is enabled to use the hardware recip instruction. Also, expressions that would have produced a NaN or an inf may produce different answers, but otherwise answers are the same as those obtained when IEEE_arithmetic=1 is in effect. Examples: 0*X may be changed to 0, and X/X may be changed to 1 even though this is inaccurate when X is +inf, -inf, or NaN. This is the **default for -O3 and -Ofast.**

3 Performs arbitrary, mathematically valid transformations, even if they can produce inaccurate results for operations specified in ANSI/IEEE 754-1985. These transformations can cause overflow or underflow for a valid operand range. An example is the conversion of x/y to x*recip(y) for MIPS IV targets. Also see the -OPT:roundoff=n option.

roundoff=n

Specifies the level of acceptable departure from source language floating-point, round-off, and overflow semantics. n can be one of the following:

n	Description
0	Inhibits optimizations that might affect the floating-point behavior. Requires strict compliance with programmed sequence, thereby inhibiting most loop optimizations. This is the default when optimization levels -O0, -O1, and -O2 are in effect.
1	Allows simple transformations that might cause limited round-off or overflow differences. Compounding such transformations could have more extensive effects.

- 2 Allows more extensive transformations, such as the reordering of reduction loops (loop unrolling).
This is the default level when -O3 is in effect.
- 3 Enables any mathematically valid transformation.
This is the default for -Ofast.

To obtain best performance in conjunction with software pipelining, specify roundoff=2 or roundoff=3. This is because reassociation is required for many transformations to break recurrences in loops. Note that the optimizations enabled by this option can rearrange expressions across parentheses or even statement boundaries. Also see the descriptions for the -OPT:IEEE_arithmetic, -OPT:fast_complex, -OPT:fast_trunc, and -OPT:fast_nint options.

• Summary of differences in the -OPT: option setting for MIPSpro.7.3.1.1m:

option	O0-O2	O3	Ofast
div_split	OFF	OFF	ON
fast_complex	OFF	OFF	ON
fast_exp	OFF	ON	ON
fast_nint	OFF	OFF	ON
fast_trunc	OFF	ON	ON
fold_intrinsics	OFF	ON	ON
fold_reassociate	OFF	ON	ON
got_call_conversion	OFF	ON	ON
IEEE_arithmetic	1	2	2
Olimit	2000	3000	2147483647
recip	OFF	ON	ON
reorg_common	OFF	ON	ON
round_off	0	2	3

Using Tuned Libraries for Optimization

One way to improve a program's performance is to link it with libraries already tuned for the target hardware.

• libm (standard math library)

The standard math library includes special **"vector intrinsics"** (i.e., vectorized version of certain functions) which takes advantage of software pipelining capabilities of R10000 to fill instruction slots in the operation.

scalar intrinsic : sin, cos, tan, asin, acos, atan, exp, log, log10, sqrt

vector instrinsic double precision : vasin, vacos, vatan, etc.

vector instrinsic single precision : vasin \mathbf{f} , vacos \mathbf{f} , vatan \mathbf{f} , etc.

- Vector and scalar routines may differ slightly; however, none of the results differ from the mathematically correct result by more than 2 ULPs.
- Loop-nest optimizer of the Fortran compilers may recognize a vector loop and **automatically** replaces it with a vector intrinsic call. You can check if these functions in your program are replaced with the vectorized version in the file.w2f.f or file.w2c.c or file.s by including -PLIST:=ON, -CLIST:=ON or -S when compile.
- O3 or -Ofast is required in order to use these vector intrinsics.
- lm is automatically linked in by Fortran.

f90|f77 -o a.out file.f -O3 [-lm]

- lm is **not** automatically linked in by cc or CC. You also need the include file **<math.h>**.

CC|cc -o a.out file.c++ (file.c) -lm

See **man math** for more information. An example of [Vector Intrinsics](#) is given in the [Loop Nest Optimization](#) section.

• libfastm (not fully IEEE-compliant, but fast results)

libfastm.a contains faster lower-precision **scalar** versions of various routines from libm.a.

- On MIPS 4 systems, libfastm.a contains these routines:

double precision : sin, cos, tan, atan2, exp, log, pow, sqrt
single precision : sinf, cosf, tanf, expf, logf, powf, sqrtf

- Routines sin, cos, tan, and atan2 should be run with underflow and overflow floating point traps disabled (the default mode) to avoid trapping in these routines.
- f90|f77|CC|cc -o object program -lfastm [-lm]

• libcomplib.sgimath

- The library complib.sgimath contains an extensive collection of industry standard libraries and SGI internally developed libraries:

- BLAS - Basic Linear Algebra Subprograms (BLAS) and the extended BLAS (Level 2 and 3)
- EISPACK - a collection of double precision Fortran subroutines that compute the eigenvalues and eigenvectors of nine classes of matrices
- LINPACK - for linear equations and linear least squares problems
- LAPACK - a public domain library of subroutines for solving dense linear algebra problems, successor to LINPACK and EISPACK
- FFT - Fast Fourier Transforms
- Convolutions
- direct linear equation solvers for sparse symmetric linear systems of equations.

- This library is being replaced by SCSL described below. Support from SGI for this library will be continued in future releases, until noted otherwise.

f90|f77|CC|cc -o a.out program -lcomplib.sgimath -fastm

f90|f77|CC|cc -o a.out program -mp -lcomplib.sgimath_mp -fastm (for multi-threaded jobs)

- Many of the routines in complib.sgimath are available from: <http://www.netlib.org>

http://www.nas.nasa.gov/~schang/origin_opt.html

See **man complib.sgimath**, **man blas**, **man fft**, **man conv**, **man lapack**, **man solvers** for more information.

• **libscsl - SGI Cray Scientific Library (SCSL)**

- SCSL includes algorithms that are carefully coded and optimized to Silicon Graphics, Inc. hardware.
- SCSL replaces complib. However, LINPACK and EISPACK, which are included in complib, will not be supported in SCSL.
- Use **ls -l /opt/scsl/scsl** to find the default version of libscsl. If a different version is needed, use **module load scsl.xxx**.
- `f90|f77|CC|cc -o object program -lscs -lfastm [-lm]`
- `f90|f77|CC|cc -o object program -mp -lscs_mp -lfastm -lm` (for multi-threaded jobs)

See **man libscsl** for details.

Acknowledgement

Assistance from the following personnel is greatly appreciated.

- SGI analyst : Bron Nelson
 - NASA Ames : Samson Cheung
 - NASA Ames : Johnny Chang
-

Glossary

◦ **basic block**

A sequence of program statements that contains no labels and no branches. A basic block can only be executed completely (because it contains no labels, it cannot be entered other than at the top; because it contains no branches, it cannot be exited before the end), and in sequence (no internal labels or branches). Any program can be decomposed into a sequence of basic blocks. The compiler optimizes units of basic blocks. An ideal profile counts the use of each basic block.

◦ **cache trashing**

Every reference to an array element results in a cache miss due to the unfortunate alignment of arrays, i.e, they all map to the same cache location.

◦ **constant propagation**

Formal parameters which always have a particular constant value can be replaced by the constant, allowing additional optimization. Global variables which are initialized to constant values and never modified can be replaced by the constant.

◦ **global name optimizations**

Global names in shared code must normally be referenced via addresses in a global table, in case they are defined or preempted by another DSO (see dso(5)). If the compiler knows it is compiling a main program and that the name is defined in another of the source files comprising the main program, an absolute reference can be substituted, eliminating a memory reference.

◦ **inlining**

Calls to a procedure are replaced by a suitably modified copy of the called procedure's body inline, even if the callee is in a different source file.

◦ **stride-one access**

A loop over an array accesses array elements from adjacent memory addresses.